Exercises for Chapter 7

Exercise 7.1: Entity Relationships in CMP 2.0

This exercise demonstrates some basic CMP 2.0 relationships between entity beans. You will create a new application archive for each of the examples in this chapter. To facilitate easy manipulation of the EJBs you will be provided with pre-packaged *jar*s for the entity beans. In this chapter we explore the complex relationship mapping between entity beans based on CMP 2.0. In this chapter we explore the relationships with respect to the descriptor declarations rather than the Deploytool usage.

Building the application for Exercise 7.1

1. Download the *ex07-1.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\ex07_1*, referred to as *$EX07_1_HOME*.
4. Open a command prompt and type `Ant` or `Ant -buildfile build.xml`

This should create *$EXAMPLES_HOME\build\ex07_1* directory and compile the required files.

$EXAMPLES_HOME\pre-built\ex07_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.

Database schema for the Cabin EJB

This exercise will create the `CustomerBeanTable`, the `AddressBeanTable`, the `PhoneBeanTable`, the `CreditCardBeanTable` and the `CustomerBean_phoneNumbers_PhoneBean_Table` tables as part of the deployment process.

Make sure that the Cloudscape server is running when compiling the classes using *Ant*. The build script has a target **create-sequencer** that creates the required **SequenceTable** table. Readers can also do the following:

```
Cloudscape -isql
Isql> …connect to TitanDB
Isql> CREATE TABLE "SequenceTable"
   (
      "SequenceName" VARCHAR(50) NOT NULL ,
      "CurrentValue" INT NOT NULL
   );
isql> exit;
```

The code examples assume that the entity bean's Primary Key is generated in most of the cases other than the *CustomerEJB*. 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.

J2EE SDK has limitations with respect to propagating the Database Connection to the custom code in the EJB. For example the `ejbCreate()` method in CreditCardBean is as shown below:

```
public Object ejbCreate( java.sql.Date exp, String numb, String
name, String org) throws CreateException
    {
         System.out.println("ejbCreate");

         setId( getNextUniqueKey() );
          setExpirationDate(exp);
         setNumber(numb);
         setNameOnCard(name);
         setCreditOrganization(org);
         return null;
    }
```

The exercise code contains a private method `getNextUniqueKey()` that gets the next available sequence value for the *CreditCard* EJB. This call will roll back the transaction if readers use the same JDBC Datasource. J2EE SDK does not allow multiple non-XA resources as parts of the same transaction. One way around the problem is to use a different DataSource object. Using the *Deploytool* or command line J2eeAdmin utility, add a new JDBC Datasource with the binding name of **jdbc/TitanUID** to the same database.

```
$prompt>j2eeadmin -addJdbcDatasource jdbc/TitanUID
jdbc:cloudscape:rmi:TitanDB;create=true
```

You can also turn of f the non-XA optimization in the *$J2EE_HOME\config\default*.properties and use another approach to get the ID outside the transaction.

Building the Application Archive for Ex07_1

Make sure the *jdbc/TitanUID* datasource is created and is bound to the J2EE server.

1. Run the *Deploytool* by typing in at command prompt `c:\>deploytool`
2. Create a new application and call it *Titan07_1App.ear*.

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

3. Create a new EJB and give it a name EX07_1EJB. As explained in Chapter 4 create the simple *CustomerEJB* by mapping the container-managed fields.
4. Similarly create *AddressEJB*, *PhoneEJB* and *CreditCardEJB* mapping only the intrinsic attributes.

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

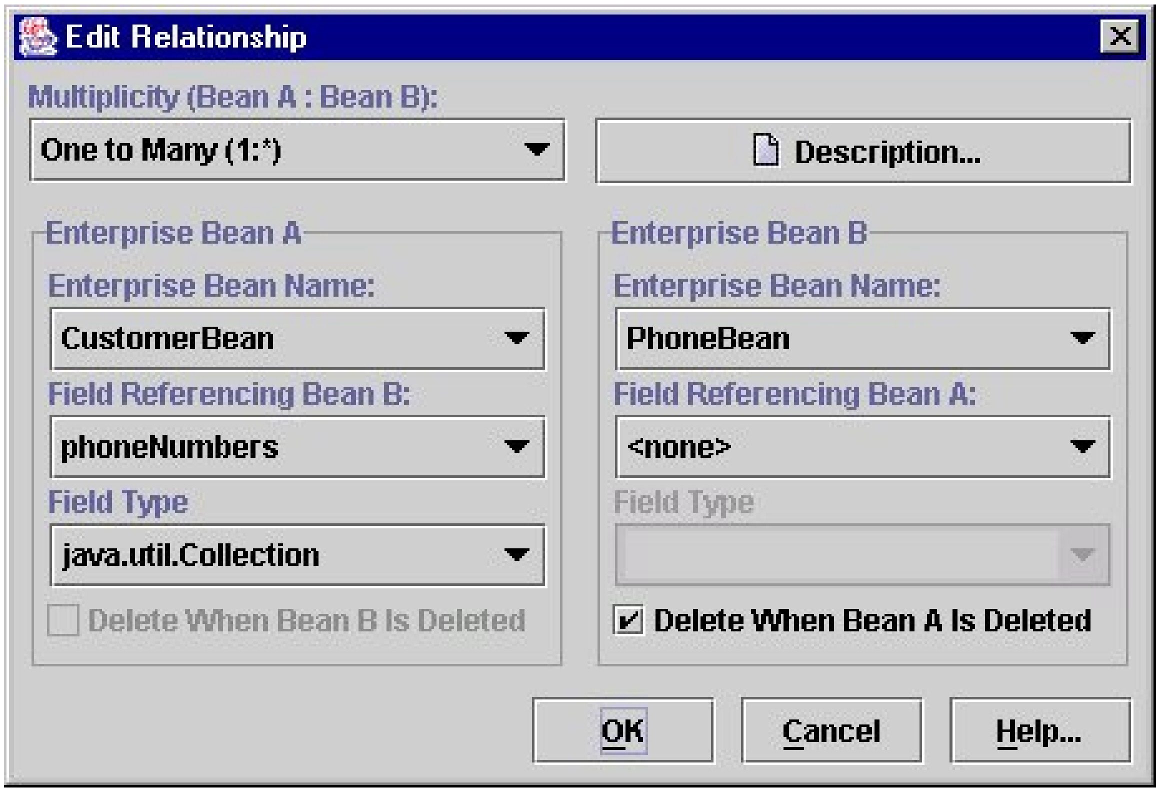
5. Select the EX07_1EJBs node and then click on the **Relationships** tab.

Before we define the relationships let us examine the relationships that exist among the entity beans in this exercise.

- Customer–Address: Customer-has-homeAddress: One-One Unidirectional
- Customer-Phone: Customer-has-many-PhoneNumbers: One-Many Unidirectional
- Customer-CreditCard: CreditCard-is-associated-to-Customer : One-One Bi-directional

6. Click the **Add** button to create relationship mappings and set them appropriately. A sample Customer-Phone Numbers relationship is shown below.

Figure 1: Customer-Phone Relationship


Edit Relationship

Multiplicity (Bean A : Bean B): One to Many (1:*)

Enterprise Bean A
Enterprise Bean Name: CustomerBean
Field Referencing Bean B: phoneNumbers
Field Type: java.util.Collection
Delete When Bean B Is Deleted [ ]

Enterprise Bean B
Enterprise Bean Name: PhoneBean
Field Referencing Bean A: <none>
Field Type:
Delete When Bean A Is Deleted [✓]

7. Select **File→Add to Application→Web War** from the menu. Select from the *$EXAMPLES_HOME/pre-built/ex06_1/Titan07_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.

8. 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** even though it shows **required**. This triggers the update of the descriptor files needed. Also select **Tools→Update Files** from the menu.

9. Deploy the application and give a web-context of titan07_1. This process will generate the required container classes and create the required tables and deploy the application.

Examine the ejb-jar descriptor file

So far in the previous chapters we have concentrated on the process of using the tool to generate the descriptors and run the example code. In this section we explore in depth the *ejb-jar xml* descriptors and the relationships between the entity beans. This workbook does not discuss the *sun-j2ee-ri.xml* descriptor file. This is J2EE™ SDK-proprietary and contains information about the persistence mapping and generation of the schema.

The *ejb-jar.xml* descriptor file contains the information about the relationships between the EJBs. Each relationship is defined in a single `<ejb-relation>` element with `<ejb-relationship-role>` elements for each of the EJBs participating in the relationship.

```
<ejb-relation>
    <description>One to One relationship showing Customer to \
       HomeAddress</description>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
         CustomerBean
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>CustomerBean</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>homeAddress</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
           AddressBean
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
```

```
          <ejb-name>AddressBean</ejb-name>
        </relationship-role-source>
      </ejb-relationship-role>
    </ejb-relation>
    <ejb-relation>
       <description>One to one relationship bi-directional between
             customer and credit card</description>
       <ejb-relationship-role>
         <ejb-relationship-role-name>
           CustomerBean
        </ejb-relationship- role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>CustomerBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>creditCard</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>
           CreditCardBean
        </ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>CreditCardBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>customer</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
   </ejb-relation>
   <ejb-relation>
      <description>One to Many relationship Uni-directional between
      customer and Phone Numbers</description>
      <ejb-relationship-role>
        <ejb-relationship-role-name>
             CustomerBean
        </ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>CustomerBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>phoneNumbers</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>
           PhoneBean
        </ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
          <ejb-name>PhoneBean</ejb-name>
        </relationship-role-source>
     </ejb-relationship-role>
</ejb-relation>
```

The *Deploytool* does not allow creating meaningful `<ejb-relationship-role-name>` element description. In general this element should contain a meaningful relationship role like *Customer-has-a-HomeAddress*.

❖ The expirationDate attribute of the CreditCardBean is changed to `java.sql.Date` rather than `java.util.Date`, since the Persistence mapping generation does not support mapping to Date in database. If `java.util.Date` is used instead the mapping will be to a different data type in the database.

This descriptor demonstrates three relationships as identified earlier. Customer-Address is a Uni-directional relationship with a multiplicity of one- one. `<cmr-field-name>` element is only defined for the source EJB. This relationship is identified by the definition of the corresponding methods in the *CustomerEJB*.

```
public abstract AddressLocal getHomeAddress();
public abstract void setHomeAddress( AddressLocal address);
```

Optionally for access to these methods these are declared in the CustomerHomeLocal interface.

```
public AddressLocal getHomeAddress();
public void setHomeAddress( AddressLocal address);
```

The Customer-CreditCard relationship is bi-directional with a multiplicity of “one”. This relationship is bi-directional and is evident in the descriptor with two `<cmr-field>` elements each for defining the relationship from the Source being *CustomerEJB* as well as *CreditCardEJB*.

The Customer-Phone EJB relationship is unidirectional and is a one-many relationship. Currently the EJB 2.0 specification supports Collection and Set types for identifying the many sides of the relationship. In this exercise the *PhoneEJB* is defined as a Collection. Existence of only one `<cmr-field>` element indicates that it is unidirectional. For each of the one-many sides of the relationship the `<ejb-relationship-role>` element identifies the relationship. For the `<relationship-role-source>` of *Customer* EJB the multiplicity is **one** and for the `<relationship-role-source>` of *Phone* EJB the multiplicity is defined as **many**.

The `addPhoneNumber(..), updatePhoneNumber(..)` and `removePhoneNumber(..)` in the *Customer* EJB provide convenient mechanisms to manipulate the Customer-Phone relationship.

All the interaction between the EJBs is through local interfaces. For this reason the return type is directly cast to the Local type.

```
public void addPhoneNumber( String number, byte type)
  throws NamingException, CreateException
{
  InitialContext jndiEnc = new InitialContext( );
  PhoneHomeLocal phoneHome =(PhoneHomeLocal)
    (jndiEnc.lookup("java:comp/env/ejb/PhoneHomeLocal"));

  PhoneLocal phone = phoneHome.create(number,type);
  Collection phoneNumbers = this.getPhoneNumbers( );
  phoneNumbers.add(phone);
}
```

Examine and Run the Client JSP Pages

This exercise consists of three JSP pages that demonstrate the relationships as discussed earlier between the *Customer*, *Address*, *Phone* and *CreditCard* EJBs. This chapter does not contain any Java client applications, and it is left as an exercise for the reader to expose the Remote interfaces for the *CustomerEJB* or through other Session EJB to interact with the Entity EJBs to test the relationships.

Client_71.jsp

The *Client_71.jsp* page demonstrates the bi-directional relationship between *Customer* and *CreditCard* EJBs. The Client JSP performs a lookup on the Home Interface and then creates a single *Customer* EJB.

```
Context jndiContext = getInitialContext();
Object obj = jndiContext.lookup("java:comp/env/CustomerHomeLocal");
CustomerHomeLocal customerhome = (CustomerHomeLocal)obj;
obj = jndiContext.lookup("java:comp/env/CreditCardHomeLocal");
CreditCardHomeLocal cardhome = (CreditCardHomeLocal)obj;
out.print("<H2>Creating Customer 71</H2>");
Integer primaryKey = new Integer(71);
CustomerLocal customer = customerhome.create(primaryKey);
customer.setName( new Name("Smith","John") );
```

Notice that the lookup is performed in such a way to obtain a reference to `javax.ejb.EJBLocalHome`. In this case it performs a direct cast to the proper type. It uses the local home object to create a `javax.ejb.EjbLocalObject`. It creates a Customer object and then it executes the business method `setName(..)`. Next it creates a CreditCardBean:

```
Calendar now = Calendar.getInstance();
```

```
java.sql.Date aDate = new java.sql.Date(System.currentTimeMillis());
CreditCardLocal card = cardhome.create(aDate , "370000000000001",
"John Smith", "O'Reilly");
out.print("<H2>Linking CreditCard and Customer</H2>");
customer.setCreditCard(card);
```

Notice that the CreditCardHomeLocal creates the CreditCard EJB using `java.sql.Date` type as one of the arguments. This is slightly different from the code in the EJB Book. J2EE SDK persistence mapping generation facilitates the mapping to Database Date datatype only if the Bean specifies `java.sql.Date` type. We generate the primary key using the `custom` method in the EJB using a different DataSource object. The client program then calls the `setCreditCard(..)` method to set up the relationship between the Customer and the CreditCard.

The following section tests the bi-directional nature of a one-one relationship between the *Customer* EJB and the *CreditCard* EJB.

```
String cardname = customer.getCreditCard().getNameOnCard();
Name name = card.getCustomer().getName();
```

Finally the client program de-references the *CreditCard* by setting the *CreditCard* reference to `null`. This removes the relationship between both the EJBs and is made evident in the output.

Open the browser and change the URL to *http://localhost:8000/titan71. Click on Client_71.jsp* and you should see the output as shown below.

```
Client_71 Example Showing Customer/CreditCard Relationship
Creating Customer 71
Creating CreditCard
Linking CreditCard and Customer
Testing both directions on relationship
customer.getCreditCard().getNameOnCard()=John Smith
card.getCustomer().getName()=John Smith
Contents of Tables
CustomerBeanTable
Obtained a database connection to Cloudscape Delete all rows in
CustomerBeanTable table
CreditCardBeanTable
Obtained a database connection to Cloudscape Delete all rows in
CreditCardBeanTable table
Unlink the beans using CreditCard, test Customer side
Card is properly unlinked from customer bean
CustomerBeanTable
Obtained a database connection to Cloudscape Delete all rows in
CustomerBeanTable table
CreditCardBeanTable
Obtained a database connection to Cloudscape Delete all rows in
CreditCardBeanTable table
```

Client_72.jsp

The *Client_72.jsp* demonstrates the relationship between the *Customer* and *Address* EJBs. As mentioned earlier this is a unidirectional relationship with a multiplicity of one. Customer has a reference to Address. *Customer* EJB has methods that define `getHomeAddress`() and `setHomeAddress()`. This client program performs multiple tests with the Customer's address. In this relationship the Address is created without a reference to Customer. Then the Address is linked to Customer. This example also demonstrates the manipulation of the fields of the Address EJB and the AddressEJB relationship itself.

This is a one-one relationship. So a `setHomeAddress()` on the customer de-references the previous address and assigns a new Address to the *CustomerEJB*. On the other hand modification to the value of the Address bean only modifies the intrinsic attribute of the *Address* EJB.

```
Client_72 Example Showing Customer/Address Relationship
Finding Customer 71
Address reference is NULL, Creating one and setting in Customer..
Address Info: 333 North Washington Minneapolis, MN 55401
Modifying Address through address reference
Address Info: 445 East Lake Street Wayzata, MN 55432
Creating New Address and calling setHomeAddress
Address Info: 700 Main Street St. Paul, MN 55302
Retrieving Address reference from Customer via getHomeAddress
Address Info: 700 Main Street St. Paul, MN 55302
```

Client_73.jsp

This client JSP page demonstrates the relationship between Customer and Phone EJB. Customer has many Phone numbers. *Customer* EJB has a *cmr* field called `phoneNumbers`. The client JSP uses the convenience methods provided on the *Customer* EJB to add, update and remove a Phone Number reference to the *Customer* EJB. This also is a uni-directional relationship.

Also note the container does not allow manipulation of the relationship collection. Explicit transaction demarcation is needed to perform operations on collections. The `addPhoneNumber` on the other hand does not need explicit transaction around the call from the client JSP. The method is a business method and the transactional semantics are defined declaratively on the method in the descriptor. Client executing `addPhoneNumber()` is already in the context of a transaction and can perform the related operations on the collection.

```
Client_73 Example Showing Customer/Phone Relationship
Starting contents of phone list:
Adding a new type 1 phone number..
New contents of phone list:
Type=1 Number=612-555-1212
Adding a new type 2 phone number..
New contents of phone list:
```

```
Type=2 Number=800-333-3333
Type=1 Number=612-555-1212
Updating type 1 phone numbers..
New contents of phone list:
Type=2 Number=800-333-3333
Type=1 Number=763-555-1212
Removing type 1 phone numbers from this Customer..
Final contents of phone list:
Type=2 Number=800-333-3333
```

This completes the simple relationship management between *Customer*, *Address*, *CreditCard* and *Phone* EJBs.

Exercise 7.2: Entity Relationships in CMP 2.0: Part 2

This exercise further demonstrates a few more relationships between other entity beans in the Titan Cruise Application. The following are the relationships that were covered in Exercise 7.1:

- One-to-One unidirectional: Customer-Address (Customer-has-homeAddress)
- One-to-One bi-directional: Customer-CreditCard (Customer-CreditCard relationship)
- One-to-Many unidirectional: Customer-Phone (Customer-has-many PhoneNumbers)

The following relationships are covered in this extended exercise:

- Many-to-One unidirectional: Cruise-Ship (Many Cruises –can-be-organized-on-a-Ship)
- One-to-Many Bi-directional: Cruise-Reservation (Each-Cruise-has-many-Reservations)
- Many-to-Many bi-directional: Customer-Reservation (Customers-many-Reservations)
- Many-to-Many unidirectional: Cabin-Reservation (Reservation-many-Cabins)

Building the application for Exercise 7.2

1. Download the *ex07-2.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\ex07_2*, referred to as *$EX07_2_HOME*.
4. Open a command prompt and type `Ant` or `Ant -buildfile build.xml`

This should create *$EXAMPLES_HOME\build\ex07_2* directory and compile the required files.

$EXAMPLES_HOME\pre-built\ex07_2 directory contains pre-packaged client application *jars* for this exercise.

5. As shown in the exercise 7.1 create the *Titan07_2App* application components.

Database schema for the Ex07_2

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

Building the Application Archive for Ex07_2

Run the *Deploytool* by typing in at command prompt `c:\>deploytool`. Create a new application and call it *Titan7_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.

1. Select any EJB and click on the **Entity** tab. Make sure the database connection parameters are set and generate the default SQL.
2. 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.**
3. Open the browser and go to *http://localhost:8000/titan72a*.

We are almost ready to run the example. But not yet! Let us explore the application components and understand them better.

Examine the ejb-jar descriptor file

The *ejb-jar.xml* descriptor file contains the information about the relationships between the EJBs. Each relationship is defined in a single `<ejb-relation>` element with `<ejb-relationship-role>` elements for each of the EJBs participating in the relationship.

Many-to-One Unidirectional: Cruise-Ship (Many Cruises –can-be-organized-on-a-Ship)

```
<ejb-relation>
         <description>
            many-cruises-associated-with-ship
         </description>
         <ejb-relationship-role>
               <ejb-relationship-role-name>
                     CruiseBean
               </ejb-relationship-role-name>
               <multiplicity>Many</multiplicity>
               <relationship-role-source>
```

```
                        <ejb-name>CruiseBean</ejb-name>
                  </relationship-role-source>
                  <cmr-field>
                        <cmr-field-name>ship</cmr-field-name>
                  </cmr-field>
            </ejb-relationship-role>
            <ejb-relationship-role>
                  <ejb-relationship-role-name>
                        ShipBean
                  </ejb-relationship-role-name>
                  <multiplicity>One</multiplicity>
                  <relationship-role-source>
                        <ejb-name>ShipBean</ejb-name>
                  </relationship-role-source>
            </ejb-relationship-role>
   </ejb-relation>
```

Cruise to Ship relationship is many-to-one. The multiplicity on the Cruise side is Many and on the Ship is one. This is a unidirectional relationship. This is implicit by one `<cmr-field>` element that defines the relationship from *CruiseBean*. This relationship is enabled by the definitions of the following methods in the *CruiseBean* EJB.

```
public abstract void setShip(ShipLocal ship);
public abstract ShipLocal getShip( );
```

The Local interface CruiseLocal also declares the following methods to enable access to the associated Ship. Also ShipBean does not define any methods to relate to the list of Cruises associated to the Ship. This demonstrates the unidirectional relationship.

```
public ShipLocal getShip();
public void setShip(ShipLocal ship);
```

The J2EE™ SDK generates the mapping for the Entity Beans with One-One for each Entity Bean and a relationship table for each relationship. Even though we could just have `ShipID` as a foreign key in the `CruiseBeanTable`, the RI mapping generates a different table for the relationship. The following is the Cloudscape schema definition for the three tables participating in the relationship.

```
CREATE TABLE "CruiseBeanTable"
   (
      "id" INT NOT NULL ,
      "name" VARCHAR(255)
   )
CREATE TABLE "ShipBeanTable"
   (
      "id" INT NOT NULL ,
      "name" VARCHAR(255),
      "tonnage" DOUBLE PRECISION NOT NULL
   )
CREATE TABLE "CruiseBean_ship_ShipBean_Table"
```

```
    (
       "_CruiseBean_id" INT NOT NULL ,
       "_ShipBean_id" INT
    )
```

- Developers working with J2EE™ SDK need to understand the limitations of the tool. Changing the mapping in the generated SQL or in the *sun-j2ee-ri.xml* doesn't help. At run time the classes generated use these semantics.

One-to-Many bidirectional: Cruise-Reservation (Each-Cruise-has-many-Reservations)

```
<ejb-relation>
   <description>Cruise-has-many-reservations</description>
   <ejb-relationship-role>
         <ejb-relationship-role-name>
              CruiseBean
         </ejb-relationship-role-name>
         <multiplicity>One</multiplicity>
         <relationship-role-source>
               <ejb-name>CruiseBean</ejb-name>
         </relationship-role-source>
         <cmr-field>
               <cmr-field-name>reservations</cmr-field-name>
               <cmr-field-type>java.util.Collection</cmr-field-type>
         </cmr-field>
   </ejb-relationship-role>
   <ejb-relationship-role>
         <ejb-relationship-role-name>
              ReservationBean
         </ejb-relationship-role-name>
         <multiplicity>Many</multiplicity>
         <relationship-role-source>
               <ejb-name>ReservationBean</ejb-name>
         </relationship-role-source>
         <cmr-field>
               <cmr-field-name>cruise</cmr-field-name>
         </cmr-field>
   </ejb-relationship-role>
</ejb-relation>
```

This is a bi-directional relationship between *CruiseBean* and *ReservationBean* EJBs. This descriptor defines `<cmr-field>` elements for both sides of the relationship. This is One-to-Many from *CruiseBean* to *ReservationBean* EJBs. The *CruiseBean* side of the relationship shows the `<cmr-field>` element as reservations a `java.util.Collection` type and the relationship from *ReservationBean* EJB being a source is defined for having a reference to the cruise. Both

EJBs define the corresponding `get` and `set` methods for each side of the relationship. *CruiseEJB* defines the following methods in the Bean class.

```
public abstract void setReservations(Collection res);
public abstract Collection getReservations( );
```

The corresponding method definitions on the `CruiseLocal` interface are:

```
public void setReservations(Collection res);
public Collection getReservations( );
```

The *ReservationBean* defines the following methods in the Bean class:

```
public abstract CruiseLocal getCruise();
 public abstract void setCruise(CruiseLocal cruise);
```

The corresponding method definitions on the `ReservationLocal` interface are:

```
public CruiseLocal getCruise();
public void setCruise(CruiseLocal cruise);
```

The database table generated for the relationship between *CruiseBean* and *ReservationBean* is as follows:

```
CREATE TABLE "CruiseBean_reservations_ReservationBean_cruiseTable"
   (
      "_CruiseBean_id" INT,
      "_ReservationBean_id" INT NOT NULL
   )
```

Many-to-Many Bi-directional: Customer-Reservation (Customers-many-Reservations)

```
<ejb-relation>
   <ejb-relationship-role>
         <ejb-relationship-role-name>
               CustomerBean
         </ejb-relationship-role-name>
         <multiplicity>Many</multiplicity>
         <relationship-role-source>
               <ejb-name>CustomerBean</ejb-name>
         </relationship-role-source>
         <cmr-field>
               <cmr-field-name>reservations</cmr-field-name>
               <cmr-field-type>java.util.Collection</cmr-field-type>
         </cmr-field>
   </ejb-relationship-role>
   <ejb-relationship-role>
         <ejb-relationship-role-name>
               ReservationBean
         </ejb-relationship-role-name>
         <multiplicity>Many</multiplicity>
         <relationship-role-source>
```

```
                <ejb-name>ReservationBean</ejb-name>
          </relationship-role-source>
          <cmr-field>
                <cmr-field-name>customers</cmr-field-name>
                <cmr-field-type>java.util.Set</cmr-field-type>
          </cmr-field>
    </ejb-relationship-role>
</ejb-relation>
```

You can see that there are two `<cmr-field>` references and the multiplicity for each side of the relationship is many. *CustomerBean* as source has many reservations and the type is defined as a `java.util.Collection;` whereas the *ReservationBean* as source has many customers and the type is defined as `java.util.Set`. In this case the relationship identifies distinct Customers so we use `java.util.Set` instead of `java.util.Collection`.

The methods in the *CustomerBean* EJB are as follows:

```
public abstract Collection getReservations();
public abstract void setReservations(Collection reservations);
```

The corresponding methods in the `CustomerLocal` interface are:

```
public Collection getReservations();
public void setReservations(Collection reservations);
```

The methods in the *ReservationBean* EJB are as follows:

```
public abstract Set getCustomers();
public abstract void setCustomers(Set customers);
```

The corresponding definitions in the `ReservationLocal` interface are:

```
public Set getCustomers( );
public void setCustomers(Set customers);
```

The ddl syntax for the table is as follows:

```
CREATE TABLE
"CustomerBean_reservations_ReservationBean_customersTable"
(
"_CustomerBean_id" INT NOT NULL ,
"_ReservationBean_id" INT NOT NULL
)
```

Many-to-Many Unidirectional: Cabin- Reservation (Reservation-many-Cabins)

```
<ejb-relation>
   <ejb-relationship-role>
          <ejb-relationship-role-name>
                ReservationBean
          </ejb-relationship-role-name>
          <multiplicity>Many</multiplicity>
          <relationship-role-source>
                <ejb-name>ReservationBean</ejb-name>
```

```
            </relationship-role-source>
            <cmr-field>
                  <cmr-field-name>cabins</cmr-field-name>
                  <cmr-field-type>java.util.Set</cmr-field-type>
            </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
            <ejb-relationship-role-name>
                  CabinBean
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                  <ejb-name>CabinBean</ejb-name>
            </relationship-role-source>
      </ejb-relationship-role>
</ejb-relation>
```

This is a many-to-many unidirectional relationship between *ReservationBean* and the *CabinBean* EJBs. *ReservationBean* has references to the *CabinBean*, but *CabinBean* does not have a reference back to *ReservationEJB*. This is indicative of only one `<cmr-field>` element that defines the set of Cabins from *ReservationBean* as source. The CabinBean side of the relationship defines the multiplicity of "many" and does not define the relationship field.

The following methods are defined in the *ReservationBean* EJB:

```
public abstract Set getCabins();
public abstract void setCabins(Set cabins);
```

The corresponding definitions in the `ReservationLocal` are as follows:

```
public Set getCabins( );
public void setCabins(Set customers);
```

The database ddl for the relationship management is as follows:

```
CREATE TABLE "ReservationBean_cabins_CabinBean_Table"
   (
      "_CabinBean_id" INT NOT NULL ,
      "_ReservationBean_id" INT NOT NULL
   )
```

❖ The CMP generation in J2EE SDK does not support more than one Many-to-One unidirectional relationship to the same Entity EJB. For example we have a Many-to-one unidirectional relationship between *CruiseEJB* and *ShipEJB* as well as *CabinEJB* to *ShipEJB*. Till deployment time the errors don't appear. It seems that the generated java classes when the application is deployed the *ShipBean_PM.class* contains re-definition of `_reverse_ship` final attribute for each relationship. One way to work around this is to make the *CabinEJB* to *ShipEJB* bi-directional. This relationship is modeled as bi-directional in Chapter 12, which is a deviation from the EJBBook.

Examine and Run the Client JSP Pages

This exercise consists of number of client JSP pages that demonstrate the relationships as explained above. It is required to undeploy Titan7_1App application and deploy Titan7_2 App application. It is also required that the Database settings for the EJB in the Deploytool be marked for **Create table on deploy** and **delete table on undeploy**. These are checkboxes that should be selected.

Client_75.jsp

This example demonstrates the Cruise-to-Ship relationship and also includes sharing the references. (See EJB Book Figure 7-12.)

These examples are self-explanatory. This JSP creates two ShipBean EJBs and six CruiseBean EJBs. It assigns the first three Cruise beans to Ship A and the last three to Ship B. It then queries the reference for the fourth Cruise bean and sets its reference to Ship A. This demonstrates the updates of the references to Ship.

```
ShipLocal newship = cruises[4].getShip();
cruises[1].setShip(newship);
```

The output for the JSP is shown below:

```
Client_75 Example Showing Cruise/Ship Relationship (Fig 7-14)
Creating Ships
PK=1001 name=Ship A tonnage=30000.0
PK=1002 name=Ship B tonnage=40000.0
Creating Cruises
Cruise 1 is using Ship A
Cruise 2 is using Ship A
Cruise 3 is using Ship A
Cruise 4 is using Ship B
Cruise 5 is using Ship B
Cruise 6 is using Ship B
Changing Cruise 1 to use same ship as Cruise 4
Cruise 1 is using Ship B
Cruise 2 is using Ship A
Cruise 3 is using Ship A
Cruise 4 is using Ship B
Cruise 5 is using Ship B
Cruise 6 is using Ship B
…………………………………….
```

To rerun the example you must delete the rows in the tables or else you will notice Duplicate-Key exceptions. To continue to the next JSP simply click the **Back** browser button to go to the listing of the JSPs.

Client_76a.jsp

This example demonstrates the Cruise-Reservation One-to-Many Unidirectional relationship. It further includes the use of set methods to modify the reservations associated with the Cruise. This example creates two Cruise Beans (Cruise A and Cruise B) and creates six Reservation Beans. Each Reservation bean sets the Cruise relationship link as part of the `create` process. Cruise references re passed to the Reservation Bean. Subsequently the example gets the Reservations from Cruise A and sets them for Cruise B. Manipulating the relationship in this fashion de-references the reservations four through six and assigns reservations one through three to Cruise B. The following code in the JSP demonstrates this step.

```
try
 {
   tran.begin();
     // show the effect of a simple "setReservations" on a cruise
     Collection a_reservations = cruiseA.getReservations();
     // cruiseB = cruisehome.findByPrimaryKey(cruiseBID);
     cruiseB.setReservations(a_reservations );
  tran.commit();
}
catch (Exception e)
{
   e.printStackTrace();
   tran.rollback();
}
```

Notice that the manipulation of the collection references is performed as part of a transaction. This is required so that the container can perform the operation on the collection as an atomic unit of work.

- In J2EE™ SDK, if these operations are not part of the transaction, the container raises the following exception message: **"the collection may have been accessed after transaction completion".**

This example can be run multiple times as the CruiseEJB and the ReservationEJB use the jdbc/TitanUID datasource to get a sequencer generated ID.

The output of the run is shown below:

```
Client_76 Example Showing Cruise/Reservation Relationship Using
set() (Fig 7-16)
Creating Cruises
name=Cruise A
name=Cruise B
Creating Reservations
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A
Reservation date=11/15/2002 amount paid= 6000.0 is for Cruise A
Reservation date=11/22/2002 amount paid= 7000.0 is for Cruise B
Reservation date=11/29/2002 amount paid= 8000.0 is for Cruise B
Reservation date=12/06/2002 amount paid= 9000.0 is for Cruise B
Testing CruiseB.setReservations( CruiseA.getReservations() )
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise B
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise B
Reservation date=11/15/2002 amount paid= 6000.0 is for Cruise B
Reservation date=11/22/2002 amount paid= 7000.0 is for No Cruise!
Reservation date=11/29/2002 amount paid= 8000.0 is for No Cruise!
Reservation date=12/06/2002 amount paid= 9000.0 is for No Cruise!
………………………………………………………
```

Client_76b.jsp

This JSP demonstrates the same relationship between Cruise and Reservation EJBs with use of `addAll()` method to modify the Reservations associated with the Cruise. (See EJB Book figure 7-15.)

In this example too we use a transactional context to perform the operations and manipulations on collections. The following code obtains the reservations from Cruise A and Cruise B. Then it executes the method `addAll(Cruise A reservations)` on Cruise B reservations. The `addAll()` method available on the Collection interface adds the reservations from Cruise A to Cruise B. The relationship between these two EJBs is one-many between Cruise and Reservation. To that effect the underlying persistence mechanism de-references Cruise A reservations and assigns them to Cruise B.

```
UserTransaction tran =
   (UserTransaction)jndiContext.lookup("java:comp/UserTransaction");
try {
         tran.begin();
         Collection a_reservations = cruiseA.getReservations();
         Collection b_reservations = cruiseB.getReservations();
         b_reservations.addAll(a_reservations);
         tran.commit();
   }
   catch (Exception e)
   {
         e.printStackTrace();
         tran.rollback();
   }
```

The output of the JSP is shown below:

```
Client_76b Example Showing Cruise/Reservation Relationship Using
addAll() (Fig 7-17)
Creating Cruises
name=Cruise A
name=Cruise B
```

```
Creating Reservations
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A
Reservation date=11/15/2002 amount paid= 6000.0 is for Cruise A
Reservation date=11/22/2002 amount paid= 7000.0 is for Cruise B
Reservation date=11/29/2002 amount paid= 8000.0 is for Cruise B
Reservation date=12/06/2002 amount paid= 9000.0 is for Cruise B
Testing using b_res.addAll(a_res) to combine reservations
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A
Reservation date=11/15/2002 amount paid= 6000.0 is for Cruise A
Reservation date=11/22/2002 amount paid= 7000.0 is for Cruise B
Reservation date=11/29/2002 amount paid= 8000.0 is for Cruise B
Reservation date=12/06/2002 amount paid= 9000.0 is for Cruise B
```

Client_77a.jsp

This example demonstrates the use of `addAll()` method for the Customer to Reservation relationship. Customer–Reservation is a many-many relationship. Manipulation of the relationship in this manner allows the customers to be shared by both the reservations. The following code in the JSP is also performed as part of a `UserTransaction`.

```
   try
   {
         tran.begin();
         Set customers_a = reservations[1].getCustomers();
         Set customers_b = reservations[2].getCustomers();
         customers_a.addAll(customers_b);
         tran.commit();
   }
   catch (Exception e)
   {
         e.printStackTrace();
         tran.rollback();
   }
```

The output of the example is as follows:

```
Client_77a Example Showing Reservation/Customer Relationships (Fig
7-17)
Creating a Ship and Cruise
cruise.getName()=Cruise A
ship.getName()=Ship A
cruise.getShip().getName()=Ship A
Creating Customers 1-6
Customer 1
Customer 2
Customer 3
Customer 4
Customer 5
Customer 6
Creating Reservations 1 and 2, each with 3 customers
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A with
customers Customer 1 Customer 2 Customer 3
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A with
customers Customer 6 Customer 5 Customer 4
Performing customers_a.addAll(customers_b) test
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A with
customers Customer 6 Customer 1 Customer 2 Customer 3 Customer 4
Customer 5
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A with
customers Customer 6 Customer 5 Customer 4
Contents of Tables
Delete All Rows in All Workbook Tables
CustomerBeanTable
Table name is :: CustomerBeanTable cloumn count is :: 6 string not
null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=1
lastName=Customer 1 Delete Row
string not null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=2
lastName=Customer 2 Delete Row
string not null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=3
lastName=Customer 3 Delete Row
string not null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=4
lastName=Customer 4 Delete Row
string not null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=5
lastName=Customer 5 Delete Row
string not null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=6
lastName=Customer 6 Delete Row

Delete all rows in CustomerBeanTable table
ReservationBeanTable
Table name is :: ReservationBeanTable cloumn count is :: 3 string
not null for delete link creation amountPaid=4000.0 date=2002-11-01
id=234 Delete Row
string not null for delete link creation amountPaid=5000.0
date=2002-11-08 id=235 Delete Row
string not null for delete link creation amountPaid=6000.0
date=2002-11-15 id=236 Delete Row
```

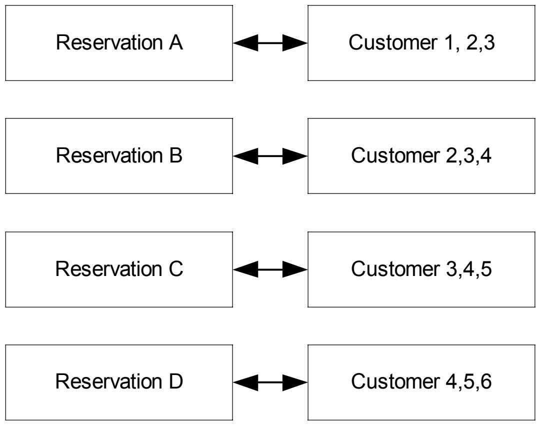
```
string not null for delete link creation amountPaid=7000.0
date=2002-11-22 id=237 Delete Row
string not null for delete link creation amountPaid=8000.0
date=2002-11-29 id=238 Delete Row
string not null for delete link creation amountPaid=9000.0
date=2002-12-06 id=239 Delete Row
string not null for delete link creation amountPaid=4000.0
date=2002-11-01 id=240 Delete Row
string not null for delete link creation amountPaid=5000.0
date=2002-11-08 id=241 Delete Row
string not null for delete link creation amountPaid=6000.0
date=2002-11-15 id=242 Delete Row
string not null for delete link creation amountPaid=7000.0
date=2002-11-22 id=243 Delete Row
string not null for delete link creation amountPaid=8000.0
date=2002-11-29 id=244 Delete Row
string not null for delete link creation amountPaid=9000.0
date=2002-12-06 id=245 Delete Row
string not null for delete link creation amountPaid=4000.0
date=2002-11-01 id=246 Delete Row
string not null for delete link creation amountPaid=5000.0
date=2002-11-08 id=247 Delete Row

Delete all rows in ReservationBeanTable table
```

- Do not re-run the example since the Customer Primary Keys are coded in the JSP. The data in the tables needs to be removed before re-running the example.

Client_77b.jsp

This JSP demonstrates the Customer-Reservations relationship using `setCustomers()` to modify the customer relationship with Reservation. This example creates six Customer EJBs and four Reservation EJBs and the relationships are set as shown in the picture.

Figure 2: Initial Relationships in Client_77b example

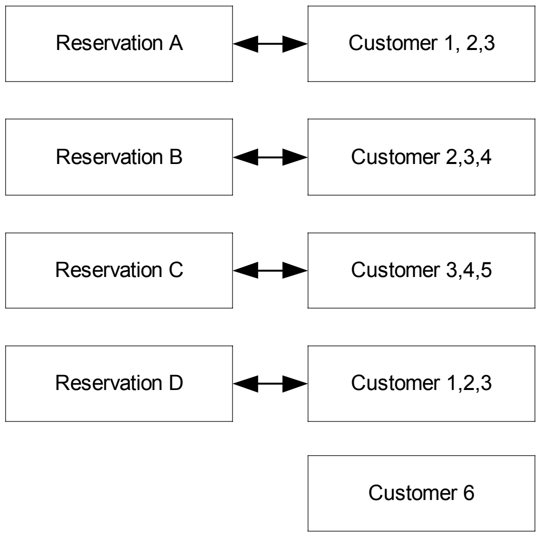


The following section manipulates these relationships using the `setCustomers` methods on the fourth reservation. The Customers from Reservation A are set on the Reservation D. Since this is a bi-directional many-many relationship, the old references for Reservation A still exist and the references for Reservation D are modified to mimic Reservation A.

```
      try
      {
            tran.begin();
            Set customers_a = reservations[1].getCustomers();
            reservations[4].setCustomers(customers_a);
            tran.commit();
      }
      catch (Exception e)
      {
            e.printStackTrace();
            tran.rollback();
      }
```

This example cannot be re-run and the tables need to be cleaned up for the next JSP to work. The following is the outcome of the manipulation of the relationships.

Figure 3: Modified relationships between Customers and Reservations



The output of the program is shown below:

```
Client_77b Example Showing Reservation/Customer Relationships (Fig
7-18)
Creating a Ship and Cruise
cruise.getName()=Cruise A
ship.getName()=Ship A
cruise.getShip().getName()=Ship A
Creating Customers 1-6
Customer 1
Customer 2
Customer 3
Customer 4
Customer 5
Customer 6
Creating Reservations 1-4 using three customers each
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A with
customers Customer 2 Customer 3 Customer 1
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A with
customers Customer 4 Customer 3 Customer 2
Reservation date=11/15/2002 amount paid= 6000.0 is for Cruise A with
customers Customer 3 Customer 4 Customer 5
Reservation date=11/22/2002 amount paid= 7000.0 is for Cruise A with
customers Customer 6 Customer 5 Customer 4
Performing reservationD.setCustomers(customersA) test
```

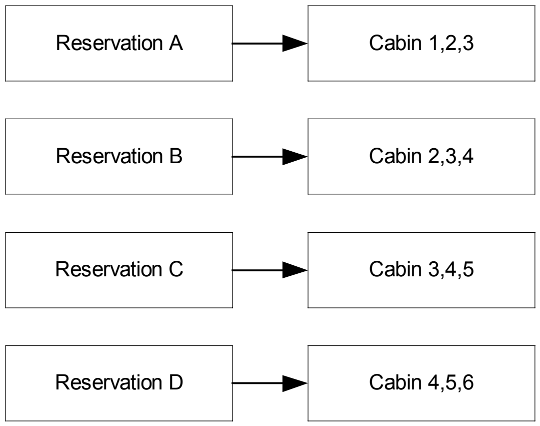
```
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A with
customers Customer 2 Customer 3 Customer 1
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A with
customers Customer 4 Customer 3 Customer 2
Reservation date=11/15/2002 amount paid= 6000.0 is for Cruise A with
customers Customer 3 Customer 4 Customer 5
Reservation date=11/22/2002 amount paid= 7000.0 is for Cruise A with
customers Customer 1 Customer 3 Customer 2
Contents of Tables
Delete All Rows in All Workbook Tables
CustomerBeanTable
Table name is :: CustomerBeanTable cloumn count is :: 6 string not
null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=1
lastName=Customer 1 Delete Row
string not null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=2
lastName=Customer 2 Delete Row
string not null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=3
lastName=Customer 3 Delete Row
string not null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=4
lastName=Customer 4 Delete Row
string not null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=5
lastName=Customer 5 Delete Row
string not null for delete link creation _creditCard_id=NULL
_homeAddress_id=NULL firstName= hasGoodCredit=false id=6
lastName=Customer 6 Delete Row

Delete all rows in CustomerBeanTable table
ReservationBeanTable
Table name is :: ReservationBeanTable cloumn count is :: 3 string
not null for delete link creation amountPaid=4000.0 date=2002-11-01
id=248 Delete Row
string not null for delete link creation amountPaid=5000.0
date=2002-11-08 id=249 Delete Row
string not null for delete link creation amountPaid=6000.0
date=2002-11-15 id=250 Delete Row
string not null for delete link creation amountPaid=7000.0
date=2002-11-22 id=251 Delete Row

Delete all rows in ReservationBeanTable table
…………………….
```

Client_77c.jsp

This JSP demonstrates the Cabin-Reservation relationship. It also demonstrates the removal of Cabins from Reservations using an Iterator. (See EJB Book Figure 7-20.)

This example creates a set of four Reservations and 6 Cabins and relates them in the same way as the previous example. The relationships are as shown below.

Figure 4 Initial Cabin to Reservation Relationship



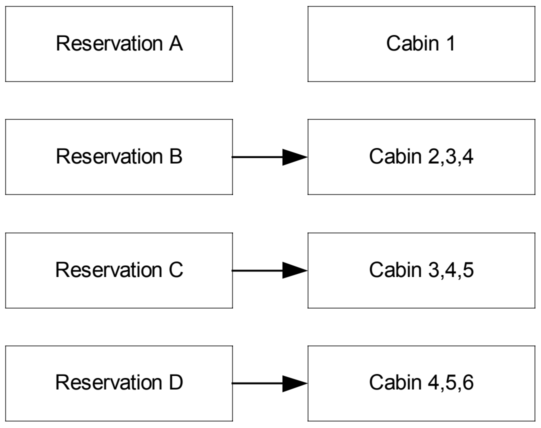
Cabin to Reservation is a unidirectional many-many relationship. This example iterates through the collection of Cabins for Reservation A and calls `remove()` on the iterator. This operation removes the relationship between Reservation A and the Cabins. As mentioned earlier these operations are performed in the context of a Transaction.

The code in the JSP performing the action is shown below:

```
try
{
   tran.begin();
   Set cabins_a = reservations[1].getCabins();
   Iterator iterator = cabins_a.iterator();
   while (iterator.hasNext())
   {
         CabinLocal cc = (CabinLocal)iterator.next();
         System.out.println("Removing "+cc.getName()+" from
           cabins_a");
         out.print("Removing "+cc.getName()+" from
           cabins_a"+"<br>");
         iterator.remove();
   }

   tran.commit();
}
catch (Exception e)
{
   e.printStackTrace();
   tran.rollback();
}
```

This results in removing the link between the CabinA and Reservations. Again this example cannot be re-run as the Cabin EJB primary keys are not generated using Sequencer.

Figure 5Removal of Cabins for Reservation A



The output is shown below:

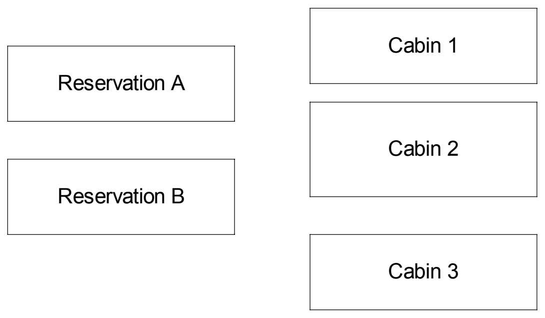
```
Client_77c Example Showing Reservation/Cabin Relationships (Fig 7-
20)
Creating a Ship and Cruise
cruise.getName()=Cruise A
ship.getName()=Ship A
cruise.getShip().getName()=Ship A
Creating Cabins 1-6
Cabin 1
Cabin 2
Cabin 3
Cabin 4
Cabin 5
Cabin 6
Creating Reservations 1-4 using three cabins each
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A with
cabins Cabin 2 Cabin 3 Cabin 1
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A with
```

```
cabins Cabin 4 Cabin 3 Cabin 2
Reservation date=11/15/2002 amount paid= 6000.0 is for Cruise A with
cabins Cabin 4 Cabin 5 Cabin 3
Reservation date=11/22/2002 amount paid= 7000.0 is for Cruise A with
cabins Cabin 5 Cabin 4 Cabin 6
Performing cabins_a collection iterator.remove() test
Removing Cabin 2 from cabins_a
Removing Cabin 3 from cabins_a
Removing Cabin 1 from cabins_a
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A with
cabins
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A with
cabins Cabin 4 Cabin 3 Cabin 2
Reservation date=11/15/2002 amount paid= 6000.0 is for Cruise A with
cabins Cabin 4 Cabin 5 Cabin 3
Reservation date=11/22/2002 amount paid= 7000.0 is for Cruise A with
cabins Cabin 5 Cabin 4 Cabin 6
```

Client_77d.jsp

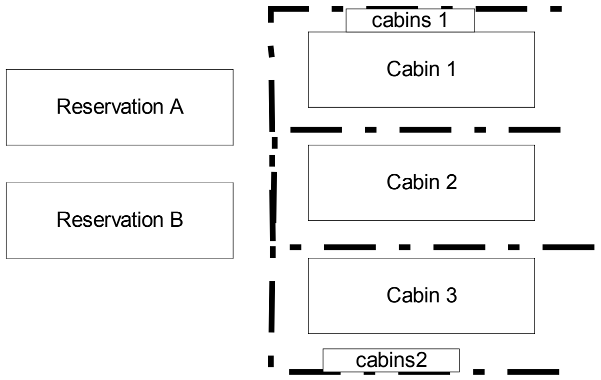
This example uses the same Cabin-Reservation relationship to demonstrate the usage of `setCabins()` to modify the cabins for a Reservation. The example first creates two Reservations and three Cabins. The relationships are not yet established. The following figure shows the relationships at this point.

Figure 6: Initial creation of Cabin and Reservations



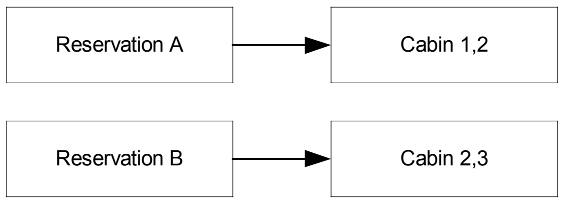
Then the example creates two new `HashSets` each containing two cabins as shown below.

Figure 7 Relationships of the created HashSets.



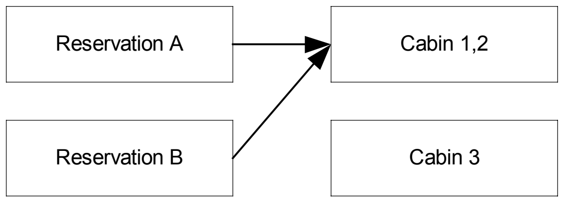
In this example the JSP then calls the `setCabins` on the reservations passing in the creates HashSets. This establishes the relationships as shown below.

Figure 8 Relationships after calling setCabins() on Reservation EJB



Then the example modifies the relationships by calling `getCabins` on Reservation A and setting them on Reservation B. Basically this de-references Cabin 3 and establishes the relationships similar to Reservation A.

Figure 9After Modifying the Cabins relationship for Reservation B



The output is shown below:

```
Client_77d Example Showing Cabin/Reservation Relationships
Creating a Ship and Cruise
cruise.getName()=Cruise A
ship.getName()=Ship A
cruise.getShip().getName()=Ship A
```

```
Creating Cabins
Minnesota Suite
California Suite
Missouri Suite
Creating Reservations
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A with
cabininfo
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A with
cabininfo
Creating Links from Reservations to Cabins
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A with
cabininfo Minnesota Suite California Suite
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A with
cabininfo Missouri Suite California Suite
Testing reservation_b.setCabins(reservation_a.getCabins())
Reservation date=11/01/2002 amount paid= 4000.0 is for Cruise A with
cabininfo Minnesota Suite California Suite
Reservation date=11/08/2002 amount paid= 5000.0 is for Cruise A with
cabininfo Minnesota Suite California Suite
```

Exercise 7.3: Cascade Deletes in CMP 2.0

This exercise, even though it is specified separately, has no additional work to be done. This exercise demonstrates the automatic cascade-delete functionality between CustomerBean and some related parts (persistent aggregates).

Downloading and building the example

This exercise is bundled into the previous exercise. The only addition specific to this exercise is *Client_78.jsp*. No additional work is needed to build the application.

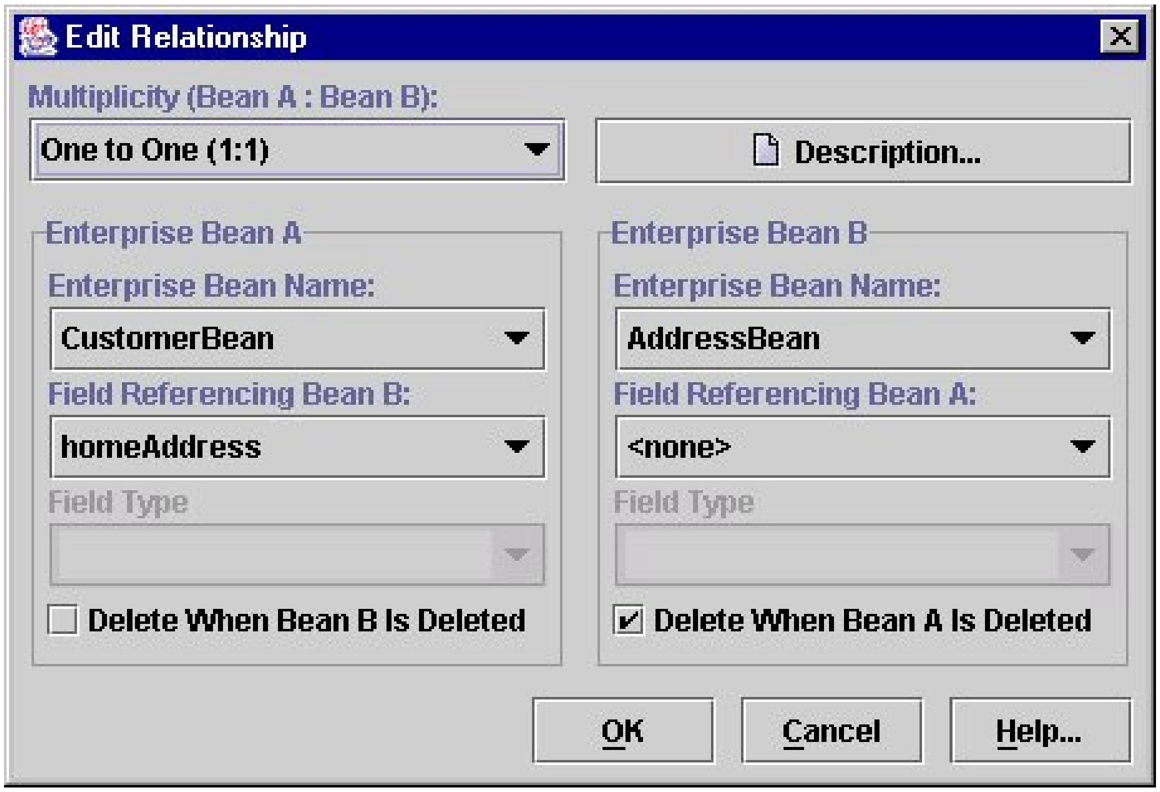
Database schema for the Ex07_3

There are no additional tables needed for this exercise. Make sure that all the data in the database is removed before running the client program.

Examine the standard ejb-jar.xml

In the previous example we purposefully did not show the cascade-delete information in the *ejb-jar.xml* descriptor file. Cascade-delete functionality is provided by placing the special `<cascade-delete/>` empty element in the relationship section of the standard *ejb-jar.xml* within the role section of the EJB that should be deleted when the parent is deleted. In this example we mark the Address to be deleted when the Customer is deleted.

Figure 10: Defining Cascade delete in Deploytool



Edit Relationship

Multiplicity (Bean A : Bean B): One to One (1:1)

Enterprise Bean A
Enterprise Bean Name: CustomerBean
Field Referencing Bean B: homeAddress
Field Type:
[ ] Delete When Bean B Is Deleted

Enterprise Bean B
Enterprise Bean Name: AddressBean
Field Referencing Bean A: <none>
Field Type:
[✓] Delete When Bean A Is Deleted

The flowing section in the *ejb-jar.xml* file demonstrates the modeling of cascade-delete functionality within the CMP 2.0.

```
<ejb-relation>
         <ejb-relationship-role>
               <ejb-relationship-role-name>
                           CustomerBean
               </ejb-relationship-role-name>
               <multiplicity>One</multiplicity>
               <relationship-role-source>
                     <ejb-name>CustomerBean</ejb-name>
               </relationship-role-source>
               <cmr-field>
                     <cmr-field-name>homeAddress</cmr-field-name>
               </cmr-field>
         </ejb-relationship-role>
         <ejb-relationship-role>
               <ejb-relationship-role-name>
```

```
                        AddressBean
                  </ejb-relationship-role-name>
                  <multiplicity>One</multiplicity>
                  <cascade-delete/>
                  <relationship-role-source>
                        <ejb-name>AddressBean</ejb-name>
                  </relationship-role-source>
            </ejb-relationship-role>
</ejb-relation>
```

Note that the *<cascade-delete/>* empty element is placed on the AddressBean side of the relationship. This identifies that within the context of Customer, deleting a Customer will delete the Customer-to-Address EJB relationship.

Examine the Run of Client JSPs

There is a single JSP that we did not execute in the previous exercise. *Client_78.jsp* has been bundled into the *Titan07_2App.ear* file. This example when run deletes the rows related to the Customer and Address relationship. The output is shown below.

```
Client_78 Example Showing Cascade Delete for Customer EJB
Obtained Customer Reference
obtained Address Reference
Obtained CreditCard Reference
Creating Customer 10078, Addresses, Credit Card, Phones
Creating CreditCard
customer.getCreditCard().getName()=Ringo Star
Creating Address
Address Info: 780 Main Street Beverly Hills, CA 90210
Creating Phones
Adding a new type 1 phone number..
Adding a new type 2 phone number..
New contents of phone list:
Type=2 Number=888-555-1212
Type=1 Number=612-555-1212
……………………………………………………………..
```

This completes the Chapter 7 Exercises. Relax and take a deep breath!