dbMapper 2.0 User Guide

July 2003

This document is a users' guide for Version 2.0 of the dbMapper package. The dbMapper package implements a Data Access Object (DAO) pattern that allows an application programmer to execute the typical create, retrieve, update, and delete (CRUD) operations on a relational database without writing SQL code. The package accomplishes this by using XML configuration files that specify the object-relational (OR) mapping of Java classes together with the Java reflection API to generate the necessary SQL statements "on the fly". This greatly reduces the effort needed to program the typical CRUD operations used by an application. This package also uses the JDBC interface so that it is portable across any database that implements JDBC. This package was developed by the ONSD Software Group of NEC America.

Prepared by:
ONSD Software Group
14040 Park Center Road, Herndon, VA 20171
NEC America

Email: onsd@necam.com
Web: http://www.onsd.nec.com/software

Table of Contents

1 Introduction

The dbMapper package provides powerful functionality to Java applications that interact with a JDBC-capable relational database. By providing various interfaces and classes that implement a type of Data Access Object (DAO) pattern, the dbMapper package eliminates the need for an application to write any SQL statements to perform the typical create, retrieve, update, and delete (CRUD) operations on a database. The following list outlines some of the main features and benefits associated with the dbMapper package.

- Eliminates the need to write SQL statements to perform typical CRUD operations on a relational database.
- Works with any relational database that supports JDBC.
- Gives the user complete control over which attributes of a class are persisted.
- Supports complex attributes (i.e. data members of a class) such as attributes that are themselves objects, as well as attributes that are arrays or collections.
- Allows the user to work with simple or composite keys.
- Is designed to work well with multi-threaded applications.
- Supports transactions.
- Does not require a proliferation of new classes (as some DAO implementations do).
- Allows the user to override the default behavior of any CRUD operation for any persisted class.
- Employs various algorithms to provide high performance.
- Provides useful classes for managing database connections.

The two key components of the dbMapper package are the `DBInterface` interface and the `DefaultMapper` class. `DBInterface` is an interface that encapsulates all of the typical CRUD operations that an application might use, and presents them to the application in the object-oriented view of the Java language. The `DefaultMapper` class is a concrete implementation of this interface. Internally, the `DefaultMapper` and its supporting classes manage all of the SQL details needed to store objects in a relational database via the `DBInterface` interface.

The rest of this document provides details and examples for the dbMapper package. The main sections are:

- **Section 2, Using a DBInterface.** Provides details of the `DBInterface`. (After reading this section, you will have a good idea of what functionality is provided by the dbMapper package.)
- **Section 3, Creating a DBInterface**. Provides details on the concrete `DefaultMapper` implementation of `DBInterface` that is provided by the dbMapper package, including the formats of the data source, object-relational (OR) mapping, and mapping context XML files.
- **Section 4, Developing with dbMapper**. Provides information on installation and configuration, as well as several detailed examples of dbMapper usage.

If you wish to first see the dbMapper package in action, you can jump directly to the "Developing With dbMapper" section for instructions on how to get started, including how to run some examples.

2 Using a DBInterface

The `DBInterface` interface provides an object-oriented view of data stored in relational database tables. It encapsulates database operations such as creating, modifying, querying, and deleting objects in the relational database. In the "DBInterface Methods" section, the `DBInterface` methods are described. First, however, we discuss some concepts that relate to `DBInterface` implementations in general.

2.1 Field Types

The dbMapper package achieves the persistence of Java objects by mapping the fields (i.e. attributes, or data members) of objects into tables of a relational database. An understanding of the capabilities and limitations of the dbMapper package is closely related to the types of fields that the package is able to persist, and how they map to the relational database. This section enumerates and describes the precise set of field types that the dbMapper package is able to persist.

The first two field types to be discussed are "basic" and "nested" fields. These types are similar in the sense that they may be stored in a single column of a relational database table. The other two types of fields used by the dbMapper package are the "complex" and "complex collection" fields. These types are more complicated in the sense that multipe columns, rows, or tables may be needed to store them. The following sections provide precise definitions of these types and their relationships to a relational database.

2.1.1 Basic Field

The simplest type of field is referred to as a "basic field". A *basic field* is defined to be a field of any of the following types: `int, short, byte, char, long, float, double, String, Integer, Short, Byte, Character, Long, Float, Double.` In general, storing a basic field to a database is a relatively simple operation. For example, in the case of a relational database, a basic field can be stored in a single column of a database table. The `Person` class, which is shown below, is an example of a class whose fields are all basic fields. This class is mapped to a single table in a relational database, `person_table`. Note that the dbMapper package requires all basic fields of any particular class to map to a single relational database table.

```
class Person {
	String firstName;
	String lastName;
	int socialSecNum;
	boolean isLiving;
}

create table person_table (
	firstname  VARCHAR(64),
	lastname   VARCHAR(64),
	ssn        INTEGER,
	living     CHAR(1)
)
```

2.1.2 Nested Field

A basic field may be nested within another field of a Java object. In that case, the field is referred to as a *nested field*. For example, `address.street, address.city, address.state, address.zip.zipCode` and `address.zip.zip4Code` are nested fields of the `Person` class:

```
class Zip {
	int zipCode;
```

```
        int zip4Code;
}
class Address {
        String street;
        String city,
        String state;
        Zip zip;
}
class Person {
        String firstName;
        String lastName;
        int socialSecNum;
        boolean isLiving;
        Address address;
}
```

In the previous section, it was noted that the dbMapper package requires all basic fields to be mapped to a single relational database table. We now extend this requirement to include nested fields. In other words, the dbMapper package requires that all basic and nested fields, for any particular class, map to a single relational database table. In this example, all of the fields of the `Person` class, which are of the basic and nested types, are mapped to a single table named `person_table`:

```
create table person_table (
        firstname  VARCHAR(64),
        lastname   VARCHAR(64),
        ssn        INTEGER,
        living     CHAR(1),
        street     VARCHAR(64),
        city       VARCHAR(64),
        state      CHAR(2),
        zip        INTEGER,
        zip4       INTEGER
)
```

Note that basic and nested fields are very similar. The main reason for distinguishing them as two different field types is that they must be handled a bit differently from the programming point of view.

2.1.3 Complex Field

A field is called a *complex field* if it is not a basic or nested field and there is a one-to-one relation between this field and the class that contains it (i.e. the field is not a collection or some other class that holds many objects). A typical example is a field whose type is some user-defined class. In general, database storage of a complex field is more complicated. In a relational database for example, a complex field and the object that contains it may be stored in different tables.

In the following example, the `favoriteMovie` field is a complex field of the `Person` class. A database for this example might be designed so that `favoriteMovie` field is a reference to an entry in a `movie` table that contains `Movie` objects.

```
class Movie {
        String title;
        int year;
        String producer;
        String director;
        Person person; // Store reference to the parent person object
}
class Person {
        String firstName;
        String lastName;
```

```
        int socialSecNum;
        boolean isLiving;

        Movie favoriteMovie;
}
```

2.1.4 Complex Collection Field

A field is called a *complex collection field* if there is a one-to-many relation between this field and the class that contains it. A typical example of a complex collection field is a field that is an array or collection. In the following example, the `favoriteMovies` field is a complex collection field. Once again, the storage of this type is more complicated than the storage of a basic or nested field.

```
class Person {
        String firstName;
        String lastName;
        int socialSecNum;
        boolean isLiving;

        Movie[] favoriteMovies;
}
```

Note that the dbmapper package works with complex collection fields that are arrays or collections of complex fields. Collections and arrays of basic fields are not supported.

Note: While the `DBInterface` itself does not distinguish between these different field types, any concrete implementation of that interface certainly must deal with them. Also, note that basic fields do play a special role in the definition of the "primary key" concept used by the dbMapper package. This concept is discussed in the following section.

2.2 Key Fields and Primary Keys

For certain operations, such as locating an object in a database for an update, the concept of a key is needed. In the case of a relational database, for example, records of a table may be located by the use of a primary key, which is a set of entries from one or more columns of the database record. The dbMapper package uses an analogous definition of a primary key. Specifically, a *primary key* for a Java class is defined to be a set of fields for that class that uniquely identifies that object. Each field of this primary key is referred to as a *key field*. In keeping with the analogy of a relational database key, a key field must be a basic field, which by definition corresponds to an individual column in a database table.

The definition of a primary key for a class that uses the dbMapper package is optional. However, as mentioned above, certain operations such as updating objects, deleting or finding objects by key, etc., do require that a primary key be defined.

Another concept related to primary keys is that of a primary key class. A *primary key class* is merely a class that encapsulates all of the key fields of the primary key. Some of the `DBInterface` methods, such as `findByPrimaryKey()`, or `findAllPrimaryKeys()`, require that a primary key class be defined. In the case where a primary key consists of a single key field, there is no need for an application to define a primary key class. It is already provided by a Java class. For example, if a primary key for a class consists of a single field of type `int`, the `java.lang.Integer` class serves as the primary key class. However, for primary keys that consist of multiple key fields, it is up to the application to provide a primary key class.

2.3 Data-Source, Mappings, Mapping Contexts, and Mappers

Any implementation of `DBInterface` must have some specific information about the objects to be saved, and the database to which they are saved. The required information is provided by "mapping contexts", which consist of "data sources" and "class mappings" (or "mapping sets"). These terms are defined below:

Data Source
The information that specifies the database and the means by which it is accessed it is called a *data source*. For instance, a data source might be a class that provides access to connections on a specified database.

Class Mapping or Mapping
A *class mapping,* or simply *mapping,* provides the information that is needed by an implementation of `DBInterface` to store objects of a specific class in a database. For example, a mapping may specify the relational database table columns that are used to store fields of a class. Note that a single class may have multiple mappings. An example that motivates the use of multiple mappings is described below under the "Mapping Context" heading.

Mapping Set
A *mapping set* is simply a set of mappings (for a set of classes). Since a mapping provides information for just one specific class, a mapping set is needed when the database is used to store objects of different classes.

Mapping Context
It is important to note that a single application may want to store different instances of a particular class in different tables, or even different databases. Furthermore, it may even want to use different mappings for the same class, depending on the context in which they are being used. The concept of a mapping context is used to provide this flexibility. By definition, a *mapping context* is a combination of a data source and a mapping set. By instantiating `DBInterface` objects with different mapping contexts, context sensitive database storage and retrieval can be achieved. This concept, which is a central concept used by the dbmapper package, is further illustrated in the following example:

Mapping Context Example

Consider an application that notifies clients of new `Widget` instances by writing the new instances into the clients' respective databases. The `widget` table used by the first client only contains the two columns: `field1` and `field2`. The other client uses a new column, named `newField`, as well as the `field1` and `field2` columns. For this example, we assume that the column names in the relational databases match the field names of the `Widget` attributes.

```
class Widget {
       String field1;
       String field2;
       String newField;
}
```

Now consider that the application is using a concrete implementation of the `DBInterface` called `DefaultMapper`, which takes a mapping context as a constructor argument. In that case, the application could instantiate two instances of the `DefaultMapper`. One instance would be instantiated with a mapping context for the first client, and another would be instantiated with a mapping context for the second client. The first mapping context would contain the information needed by the `DefaultMapper` to write objects into the two columns, `field1` and `field2`, of the database belonging to the first client. The second mapping context would contain the information needed by the `DefaultMapper` to write objects into the three columns, `field1` and `field2` and `newField`, of the database belonging to the second client. The application code would look something like this (note that the classes and method

signatures used in this example are all fictional; they are only used to demonstrate the concept and utility of mapping contexts):

```
// use mapping context for client1
DBInterface dbi1 = new DefaultMapper(mappingContext1);
// use mapping context for client2
DBInterface dbi2 = new DefaultMapper(mappingContext2);
…
Widget widget = new Widget(…);
dbi1.create(widget); // write the widget to the first client's database
dbi2.create(widget); // write the widget to the second client's database
```

In general, all methods executed by a `DBInterface` object are done in the context of a mapping context. The exact information that must be included in a mapping context is defined by the concrete implementation of the `DBInterface` interface that is used. For example, the mapping context information that is needed by the `DefaultMapper` class, which is a concrete `DBInterface` implementation provided by the dbMapper package, is described in detail in the Creating a DBInterface section.

Mapper

The term *mapper* refers to any object that implements the `DBInterface` interface.

2.4 Transaction Model

All `DBInterface` operations occur within the context of a transaction. Such transactions are expected to satisfy the ACID (atomic, consistent, isolated, durable) conditions. While it is up to the concrete implementations of `DBInterface` to implement transactions, the `DBInterface` interface does provide some useful methods for modeling transactions. The following paragraph describes the basic behavior that `DBInterface` implementations are expected to follow with respect to transactions. A more complete description is provided in the "User Managed Transaction Methods" section.

By default, each single update, create, or delete method called on a `DBInterface` object is expected to occur in the context of a single transaction that is transparent to the user. However, in a situation where a user wants to execute a *set* of `DBInterface` methods as a single transaction, the `DBInterface` interface provides methods that let an application specify the start and end of a transaction. It is up to the `DBInterface` implementation to ensure the atomicity of the set of operations that are executed between the start and end of the transaction.

2.5 DBInterface Methods

All of the methods defined by `DBInterface` are provided in the following list. Subsequence sections provide explanations of the various methods and their usage. In some cases, sample code snippets are used to illustrate the simplicity of using this interface. For more detailed technical information on the `DBInterface` methods, please refer to the dbMapper javadoc API (in the "doc/javadoc" directory) and the demo code (in the "examples" directory).

`create methods`

```
create()            // write an object to the database,
                    // basic and nested attributes[1] only
createTree()        // write an entire object containment tree to the database
```

`delete methods`

```
delete()              // delete an object from the database
deleteByAttributes()  // delete objects with certain attribute values
deleteByPrimaryKey()  // delete objects with certain key values

update methods
update ()             // update an object in the database,
                      // basic and nested fields[1] only
updateTree ()         // update an entire object containment tree in the database

finder methods
findAll ()            // get all objects of a specific class from the database
findAllPrimaryKeys () // get all primary keys for a specific class
findByAttributes ()   // get all objects that match certain attribute values
findByPrimaryKey ()   // get the object for the specified key
findByQuery ()        // get a set of objects using a user-defined SQL query

findPrimaryKeysByAttributes () // get a set of keys for objects that match
                               // specified attribute values
findPrimaryKeysByQuery () // get a set of keys using a user-defined SQL query

other (custom SQL) methods
executeQuery()        // execute an SQL query, and return the result set
executeUpdate()       // execute an SQL INSERT, UPDATE, or DELETE statement

transactional methods
beginTransaction()    // begin a transaction
commitTransaction()   // commit a transaction
rollbackTransaction() // rollback (cancel) a transaction
isActiveTransaction() // determine if the current thread is executing a transaction
```

Note 1 The concept of basic and nested fields is described fully in the "Field Types" section.

2.6 Creating a User Object

The create(Object) method
This method creates a new entry in the database for the specified object. Note that this method saves only basic and nested attributes. (To include complex and complex collection fields, use the `createTree` method.) This method throws an exception if an error occurs, e.g. a primary key violation, mapping not found, etc.

The following code snippet creates a new `Point` object and persists its basic and nested fields to the database:

```
Point p = new Point (33.4, -87.9);
dbIf.create (p);
```

The createTree (Object) method
This method creates a new entry in the database for the specified object. In contrast to the `create` method, this method saves all field types, including the complex and complex collection types. As a result, a call to this method, which employs a recursive algorithm, saves the entire containment tree represented by the Object to the database. For example, a complex field of a saved object might contain another complex field, and that field itself might contain another complex field, etc.

The following example demonstrates the `createTree` method usage for a `Path` object composed of basic, complex, and complex collection fields. In this example, a `path` is made of `PathElement` objects, which are in turn made of `Point` objects. Thus, creation of a `path` object results in persistence of the entire containment tree, including all intermediate `PathElement` objects, as well as the leaf node `Point` objects.

```
Path path = new Path (pathId);
path.setCyclic(isCyclic);
PathElement elem1 = new PathElement(new Point(1, 10), curvature1);
elem1.setWidth(width1);
path.addElem (elem1);
//Create and customize PathElement elem2
.............
path.addElem (elem2);

// Persist entire Path object containment tree to database.
dbIf.createTree (path);
```

The createTree (Object,int) method

The second form of `createTree` limits the recursion depth to the value specified by the `int` parameter. For example, if the recursion depth is set to a value of zero, only the basic and nested fields of Object are persisted. Thus the following two calls are equivalent: `createTree(userObj,0)` and `create(userObj)`. If the recusion depth is set to a value of one, the basic and nested fields one level lower (i.e. the basic and nested fields of any complex or complex collection fields of Object) are saved. As a result, if the recursion depth is sufficiently large, the entire containment tree is saved. Thus calls to `createTree(userObj, aLargeNumber)` and `createTree(userObj)` are equivalent.

If the example used in the `createTree(Object)` section were modified so that were not necessary to save the `Point` objects to the database, then the following line of code could be used.

```
dbIf.createTree(path,1);
```

In this case, the `Point` objects (i.e. the fields of the `Point` objects) are not stored, since they occur at a recursion depth of two. All basic and nested fields of the `path` object, which corresponds to a depth of zero, and the `PathElement` objects, which correspond to a depth of one, are saved to the database.

2.7 Removing a User Object

This section discusses various methods that may be used to remove objects from the database. Note that removal of an object from a database includes removal of the entire containment tree represented by that object.

The delete(Object userObject) method

This method removes a specified user object from a database (if it can be found). An exception is thrown if a database error occurs. The following code snippet removes a `Person` object from a database:

```
dbIf.delete (person);
```

The deleteByPrimaryKey(Object primaryKey, Class userObjectClass) method

This method removes the user object specified by a primary key. The following example removes a `Person` object:

```
PersonKey pk = new PersonKey (firstName, middleName, lastName,
                              homePhoneNumber);
dbIf.deleteByPrimaryKey (pk, Person.class);
```

The deleteByAttributes(AttrValMap attributes, Class userObjectClass) method

This method removes all objects of the specified class whose field values match those specified in the attribute value map. Attributes specified in the attribute value map need not be key fields. The following code snippet removes all `Person` objects with last name `"Smith"` from the database:

```
AttrValMap attrValMap = new AttrValMap();
attrValMap.put ("lastName", "Smith");
dbIf.deleteByAttributes (attrValMap, Person.class);
```

2.8 Updating a User Object

The update(Object userObject) method

This method updates an existing database entry with the contents of `userObject`. The database entry is located using the primary key information stored in `userObject`. Note that this method only updates non-key basic and nested fields. To include complex and complex collection fields as well, the `updateTree` method, which is described below, must be used. An exception is thrown if an error occurs, e.g. database constraint violation, mapping not found, etc. .

The following code snippet creates a new persistent `Person` object, updates some fields (`e-mail`, `address` and `fax` number) of the object, then applies these changes to the database:

```
Person p = new Person (fistName, middleName, lastName, homePhoneNumber);
p.setEmail("someone@somewhere.com");
p.setFax("(111)222-3456");
dbIf.create(p);
p.setEmail("someone@somewhere_else.com");
p.setFax(null);
dbIf.update (p);
```

The updateTree(Object userObject) method

Similar to the the `createTree(Object)` method, this method employs a recursive algorithm to update to the entire containment tree in the database for the specified `userObject`. As in the case of the `create` method, updates will be applied recursively starting from the specified object, `userObject`, down to all leaf nodes of the object containment tree. This method saves all field types (i.e. basic, nested, complex, and complex collection fields).

The following example demonstrates use of the `updateTree` method for a `Path` object composed of basic, complex and complex collection fields:

```
Path path = .... // Create and fill Path object containment tree
dbIf.createTree (path); // Persist entire Path object containment tree
                        // to database.
path.setCyclic(false);
// Delete first PathElement from the path
path.removeElem(path.getElem(0));
// Modify an existing PathElement
PathElement elem2 = path.getElem(1);
elem2.setWidth(anotherWidth);
Point p = elem2.getPosition();
p.setY(-7);
// Add a new PathElement
PathElement elem3 = new PathElement(new Point(4, 14), curvature3);
path.addElem (elem3);

dbIf.updateTree (path); // Apply all the changes in Path object
                        // containment tree to database.
```

The updateTree(Object userObject, int) method

This method recursively updates all of the fields of the specified `userObject` within a given recursion depth, as specified by the `int` parameter. As in the case of the `create` method, the value of `int` specifies the number of levels for which the update is called. For example, a value of one will update the parent

object and all objects just below the parent object. To restrict updates to a depth of one in the above code, the following code snippet can be used:

```
dbIf.updateTree (path,1); // update Path and PathElements, but not Points
```

The update(Object userObject, AttrValMap attrValMap, boolean bUpdateUserObject) method
This method saves specified fields of a user object to the database. The key fields of `userObject` should not be modified so that the corresponding database record can be located. The `attrValMap` contains a set of attribute/value pairs that specify the field values to be updated. The `bUpdateUserObject` flag indicates whether the changes are to be applied to the user object after a successful database update. This flag can be useful in the context of a transaction, where the user may not want the values of the original object to be changed until the transaction is committed. The following example updates selected fields to the database:

```
Person p = .... // Create and fill person object containment tree
dbIf.createTree (p); // Persist person object containment tree
                     // to database.
AttrValMap attrValMap = new AttrValMap();
attrValMap.put ("email", "abc@xyz.com");
attrValMap.put ("address", new Address ("Street, #Apt", "city", "state",
                zipCode));
dbIf.update (p, attrValMap, true);
String emailAfterUpdate = p.getEmail();
// emailAfterUpdate should be set to "abc@xyz.com"
```

The update(Object userObject, HashMap attrValMap, boolean bUpdateUserObject) method
This method is identical to the previous `update` method, except that the set of attribute/value pairs is specified by a `HashMap` instead of an `AttrValMap` data structure.

```
HashMap hValMap = null;
Person p = .... // Create and fill person object containment tree
dbIf.createTree (p); // Persist person object containment tree
                     // to database.
hValMap = new HashMap(2);
hValMap.put ("email", "abc@xyz.com");
hValMap.put ("address", new Address("Street, #Apt", "city", "state",
zipCode));
dbIf.update(p, hValMap, true);
String emailAfterUpdate = p.getEmail();
// emailAfterUpdate should be set to "abc@xyz.com"
```

2.9 Finding User Objects

The findByPrimaryKey(Object primaryKey, Class userObjectClass) method
This method returns an object, populated with basic and nested fields only, corresponding to the database entry that matches the specified key and class. (To get an object with all fields populated, use the version of this method below.) An exception is thrown if an error occurs, e.g. invalid primary key, mapping not found, etc. The following code snippet locates the `Person` object specified by the primary key class:

```
PersonKey pk = new PersonKey (fistName, middleName, lastName,
                              homePhoneNumber);
Person p = dbIf.findByPrimaryKey (pk, Person.class);
```

The following code loads a `Path` object from the database by specifying a single primary key:

```
Path path = dbIf.findByPrimaryKey (new Integer(pathId), Path.class);
```

The findByPrimaryKey(Object primaryKey, Class userObjectClass,int depth) method
This method returns an object, populated up to the specified recursion depth, corresponding to the database entry that matches the specified key and class. This method can be used to load part or all of a user object containment trees. This method tries to recursively load all objects, starting from the object identified by the `primaryKey`, and terminating at the specified recursion `depth`. The following code snippet loads the entire object containment tree from database for a `path` object specified by a primary key:

```
Path path = dbIf.findByPrimaryKey (new Integer(pathId), Path.class,
                                   9999);
int w = path.getElem(0).getWidth(); // Access an object in the
                                    // containment tree
```

The findByAttributes(AttrValMap attributes, Class userObjectClass) method
This finder method returns a collection of user objects whose field values match those specified in the specified attribute value map. The attribute value map is expected to contain at least one attribute. Also all the attributes should be limited to basic fields. The `findByAttributes` method can be used in place of the `findByPrimaryKey` method by storing all of the key field values in the attribute value map. This technique is most useful for cases where a primary key class is not defined. The following example returns all of the `Person` objects with last name `"Smith"` from the database:

```
AttrValMap attrValMap = new AttrValMap();
attrValMap.put ("lastName", "Smith");
Collection smiths = dbIf.findByAttributes (attrValMap, Person.class);
```

The findAll(Class userObjectClass) method
This method retrieves all user objects that belong to the specified class. Only basic and nested fields of the objects will be retrieved when this method is used. The following piece of code loads all `Person` objects:

```
Collection people = dbIf.findAll (attrValMap, Person.class);
```

The findByQuery(String query, Class userObjectClass) method
Although the finder methods described above should fulfill the needs of most applications with regard to loading user objects from a database to memory, there may still be a need to allow the user to use custom SQL queries to retrieve a set of user objects from the database. Examples of such cases are relational table joins, sub-queries, etc. The `findByQuery` method allows execution of such custom SQL queries. The resulting user objects are returned in a `Collection` object. The SQL query is expected to be a valid JDBC query, which should return database rows that contain all of the fields defined by the mapping for this object. One should use this method only if none of the other finder methods (`findByPrimaryKey`, `findByAttributes`, `findAll`) serves the purpose. This method is the most flexible of all finder methods, but requires that SQL details be included in the user code, which is generally not desirable. The following code snippet demonstrates the usage of the `findByQuery` method:

```
// Find all the points lying within 10-unit radius from point (4, 5)
String query = "select x, y from point_table where
               (((x-4)*(x-4)+(y-5)*(y-5)) <= 100)"
Collection points = dbIf.findByQuery(query, Point.class);
```

The `findByAttributes`, `findAll`, and `findByQuery` methods only fill the basic fields of the user object(s).

2.10 Finding Primary Keys

The primary key finder methods have signatures very similar to those described in the "Finding User Objects" section.

The findPrimaryKeysByAttributes(AttrValMap attributes, Class userObjectClass) method
This method returns a collection of primary key objects whose field values match those specified in the attribute value map. The following example returns all of the `Person` primary keys for all objects in the database (in the mapping context of `dbIf`) whose last name is `"Smith"`:

```
AttrValMap attrValMap = new AttrValMap();
attrValMap.put ("lastName", "Smith");
Collection smithKeys = dbIf.findPrimaryKeysByAttributes (attrValMap,
                       Person.class); // Collection of PersonKey objects
```

The findAllPrimaryKeys (Class userObjectClass) method
This method retrieves all primary key objects belonging to the specified class.

```
Collection personKeys = dbIf.findAllPrimaryKeys (attrValMap,
                        Person.class); // Collection of PersonKey objects
```

The findPrimaryKeysByQuery(String query, Class userObjectClass) method
This method returns a collection of primary key objects based on a specified SQL query. This method should only be used if none of other primary key finder methods (`findPrimaryKeysByAttributes`, `findAllPrimaryKeys`) can serve the purpose. The SQL query is expected to return database rows that contain all of the key attributes defined by the class mapping of the specified `userObjectClass`. The following code finds all the primary key objects associated with people whose phone numbers are listed with `"Verizon"`.

```
String query = "select p.firstname, p.middlename, p.lastname, p.homephone
                from PERSON p, PHONE_COMPANY c where c.company='Verizon'
                and c.homephone=c.phonenumber and c.firstname=p.firstname
                and c.middlename=p.middlename and c.lastname=p.lastname"
Collection verizonKeys = dbIf.findPrimaryKeysByQuery(query,
                       Person.class); // Collection of PersonKey objects
```

2.11 User-managed Transaction Methods

The previous section dealt with the various atomic `DBInterface` methods, for which transactions are handled internally within the methods themselves. However, `DBInterface` also allows the application to manage transaction boundaries across a set of `DBInterface` method invocations. This allows users to create their own transactions when multiple updates need to be done as part of an atomic operation. One simple restriction placed on these transactions is that they are all executed within a single thread. Multiple threads cannot participate in the same transaction.

The beginTransaction() method
This method creates a new transaction and associates it with the current thread. After successful invocation of this method, all `DBInterface` methods called from the same thread are executed as part of this transaction. The same database resource (connection) is used for all method invocations belonging to this transaction, and the results are applied to the database only when the user explicitly terminates the transaction for the current thread with the `commitTransaction()` method.

The isActiveTransaction() method
This boolean method indicates whether the current thread is actively involved in a transaction. A value of `true` is returned if the thread is in an active transaction. Otherwise, `false` is returned.

The commitTransaction() method
This method commits all of the database changes made in the transaction associated with the current thread. When this method completes, the thread is no longer associated with a transaction. This method ensures

that all changes are applied to the database in an atomic manner. If any single update within this transaction fails (database corruption, disk space problem etc) then all the updates will be rolled back.

The rollbackTransaction() method
A call to this method rolls back all the database changes made within the current transaction (i.e. all calls made in this thread since `beginTransaction()` was called). This method can be invoked at any point of time within a transaction. When the `rollbackTransaction()`method completes, the thread is no longer associated with a transaction.

The following example demonstrates a typical transaction. This example describes the processing that occurs when a customer books a flight. The database needs to be updated with the payment information, reservation details and ticket details in a single atomic operation. Otherwise, if a `ticket` object creation were to fail after the corresponding `payment` object had already been successfully updated, this might lead to one very irate customer! By using the transaction, this code ensures that all three objects are stored or none are stored. Thus, the customer will not be charged unless his ticket is created.

```
dbIf.beginTransaction();
try {
   Reservation reservation = new Reservation (customerId, price, date);
   Payment payment = processCreditCardPayment (customerId, price,
                                               creditCardInfo);
   Ticket ticket = new Ticket (customerId, date);
   dbIf.create (reservation);
   dbIf.create (payment);
   dbIf.delete (ticket);
   dbIf.commitTransaction();
   issueTicketToCustomer(ticket);
}
catch (Exception ex) {
   ex.printStackTrace();
   dbIf.rollbackTransaction();
}
```

2.12 Other Direct Database Access Methods

Although the `DBInterface` methods handle most typical database operations that an application needs, there are some complex cases, such as relational table joins, sub-queries, etc., in which an application programmer may need to execute custom SQL queries. The `DBInterface` provides a set of methods that support such custom database operations.

The getConnection method
This method allows a user to get a direct connection to the database, so that custom queries, updates, or other custom database operations may be performed. The nature of the connection is determined by the data source of the mapper (i.e. concrete `DBInterface` implementation) that is being used. For example, a mapper may use a connection pool that is shared among several mappers as its data source. In that case, the `getConnection` method returns one of the free connections from that pool.

The releaseConnection method
This method is used to release a connection obtained by the `getConnection` method. It should be called when a connection obtained by `getConnection` is no longer in use. Note that failure to release connections obtained by the `getConnection` method may exhaust all connections that are available through `getConnection`.

The executeQuery(java.lang.String query) method

This method executes an SQL statement that returns a single `ResultSet` object and returns the `java.sql.ResultSet` object to the caller. Typically, the `query` is a static SQL SELECT statement. The user should close the returned result set before invoking another `DBInterface` method.

The executeUpdate(java.lang.String query) method
This method executes an SQL INSERT, UPDATE or DELETE statement, or an SQL statement that returns nothing. It returns either the row count for the INSERT, UPDATE or DELETE statement, or zero for SQL statements that return nothing.

The following code snippet gets a database connection, performs some operations on it, and then releases it.

```
DBConnection conn = dbIf.getConnection();
java.sql.ResultSet rs = null;
if (null != conn) {
   try {
      java.sql.Statement stmt = conn.getConnection();
      rs = stmt.executeQuery (query);
      // perform some operations on result set
      rs.close();
      int rows = stmt.executeUpdate(updateQuery);
   }
   finally {
       if (null != rs) {
           try { rs.close(); } catch(Exception innerEx) {}
        }
        // No need to close statement as its lifecycle
        // is maintained by DBInterface
        dbIf.releaseConnection();
   }
}
```

3 Creating a DBInterface

3.1 Introduction

The previous section described the `DBInterface` interface in detail, including a discussion of relevant concepts, as well as a detailed explanation of the methods provided by that interface. This section discusses the concrete implementation of `DBInterface` that is provided by the dbMapper package, as well as other supporting classes and interfaces. The two key classes of the dbMapper package are the `DBModule` and `DefaultMapper` classes. The `DefaultMapper` is a concrete implementation of the `DBInterface` interface, and the `DBModule` class is used to instantiate `DefaultMapper` instances. Use of these classes eliminates the need for the application programmer to write SQL code to perform standard database operations related to the persistence of Java objects in relational databases.

Before discussing details of the programming API and configuration files that are needed to use the dbMapper package, an overview of the key concepts and terms is provided in the following section. It may also be useful to review the "Using a DBInterface" section before reading this section.

3.2 Overview of Key Classes, Concepts, and Data

This section discusses the two key classes of the dbMapper package, `DBModule` and `DefaultMapper`, and the concepts, terminology, and data files associated with them. This section assumes that the reader is

already familiar with the concepts presented in the "Data-Source, Mappings, Mapping Contexts, and Mappers" section, such as mappings, mapping sets, mapping contexts, and mappers.

3.2.1 DefaultMapper Class

The `DefaultMapper` class is a concrete implementation of the `DBInterface` interface. It is the workhorse of the dbMapper package. Each instance of the `DefaultMapper` class, which we refer to as a "mapper", is associated with a single mapping context. Recall that a mapping context, which consists of a data source and a mapping set, specifies information that allows a `DBInterface` instance to save instances of specified Java classes to a specified relational database. The relational database and the means to access it are encapsulated in the data source. The Java classes that may be persisted, and the mapping details needed to accomplish persistence in a relational database, are encapsulated in the mapping set.

Note that two features of the DefaultMapper class are (1) the ability to manage transactions, and (2) a flexible mechanism for overriding the default behavior of any subset of DBInterface methods for any mapped class. The implementation of transactions is discussed in the "Transaction Model" and "User-managed Transaction Methods" sections, and the mechanism for overriding default behavior is discussed in the "Custom Database Processing: DAOs (Data Access Objects) and the DAOInterface" section.

3.2.2 DBModule Class

The dbMapper package provides a singleton class, `DBModule`, which initializes the dbMapper package and manages the creation of all mappers required by an application. In the following example, the dbMapper package is initialized, and two mappers are created.

```
// Initialize the dbMapper library
DBModule dbm = DBModule.init("dbmapper.xml");
// Get a DefaultMapper (database interface) for the specified mapping
context
DBInterface dbIf1 = dbm.createDefaultMapper("my_context1");
DBInterface dbIf2 = dbm.createDefaultMapper("my_context2");
```

The "dbmapper`.xml`" argument to the `init` method specifies the name of the mapper configuration file to be used by the application. A "mapper configuration file", which is created by the application programmer, includes all of the information needed to instantiate mappers, including the specification of one or more mapping contexts. These mapping contexts, which are identified by name, are passed as arguments to the `createDefaultMapper` method to create specific mappers. (Refer to the "Mapper Configuration Files" and "Creating a Mapper Configuration File" sections for a detailed description of the content and format of these files.)

As discussed above, the `init` method of the `DBModule` class takes a mapper configuration file as an argument. However, the mapper configuration file itself may refer to other supporting files: namely class mapping files and database connection files. The appropriate `init` method to be used depends on how the mapper configuration file and the supporting files are organized. The following list describes the available options.

- **init (String startingDir, String configXMLFile).** This method initializes `DBModule` with data from the mapper configuration file specified by `configXMLFile`, that is located in the "start directory" specified by `startingDir`. Any supporting files referenced by the mapper configuration file are loaded relative to the start directory. The start directory can be an absolute path to a directory or a path relative to the application run directory. If the files are to be loaded from a jar file, the start directory must refer to an absolute path (i.e. start with '/').

- **init (String configXMLFile)**. This method initializes `DBModule` with data from the mapper configuration file specified by `configXMLFile`. All of the files (including the mapper configuration file) will be loaded relative to the application run directory. This method is equivalent to `init(".",configXMLFile)`.
- **init()**. This method initializes `DBModule` without specifying any mapper configuration file. In this case, the dbMapper package is initialized without any mapping context or data sources. (When this method is used, the application programmer must supply configuration data to `DBModule` via the programming API. Refer to the Javadoc documentation for details. The programming API is currently not covered by this users guide.)

3.2.3 Mapper Configuration Files

The mapping contexts that are used by mappers, and the supporting data such as data sources and mapping sets are represented in an XML file referred to as a *mapper configuration file*. This configuration file is used by the `DBModule` class to instantiate mappers that use specified mapping contexts. A mapper configuration file also contains other settings, such as logging settings to be used by mappers. In this section, we give an overview of the mapper configuration file contents and format. For a complete specification of the file format, see the "Creating a Mapper Configuration File" section.

The main section of the mapper configuration file is delimited by the `<mapping_contexts>` tag. This section specifies one or more mapping contexts. Each mapping context, which is delimited by the `<mapping_context>` tag, contains the following information:

- **Mapping context name**. The `id` attribute specifies the mapping context name. This name is passed as an argument to the `createDefaultMapper` method of the `DBModule` class to instantiate a mapper.
- **Data source ID**. A data source ID, which is specified by the `data_source_id` attribute, specifies the data source for the mapping context. The data source itself is defined in another section of the mapper configuration file delimited by the `<data_sources>` tag.
- **Mapping set files**. An `or_mapping_files` element specifies a set of one or more files referred to as "mapping files". Each mapping file contains mappings for one or more Java classes. By default, all of the mappings of all specified mapping set files are included in the mapping context. However, specific mappings within each file may be included or excluded using include and exclude tags.

Note that the mapper configuration file format is designed so that any mapping context in the file may use any of the data sources or mapping sets defined in the file. This provides a great deal of flexibility in defining mapping contexts. More information on the data sources and mapping set files used by the dbMapper package are provided in the next two sections.

3.2.4 Data Sources and the `DataSource` Interface

One main component of a mapping context is the data source. The data source is used by a mapper to establish and manage connections (one or more) to a database.

All data sources used by the dbMapper package must implement the `DataSource` interface. Implementations of this interface are essentially database connection managers. Internally, the dbMapper package uses the `getConnection` and `releaseConnection` methods as needed to support the other `DBInterface` methods. (Note that unless the application programmer needs to execute some custom

database operations, these methods need not be called by the application code. The `getConnection` method is used to get a database connection object from the `DataSource` object. Once the connection is no longer needed, e.g. some set of database operations have been completed, the connection should be returned to the data source by invoking the `releaseConnection` method.) The dbMapper package uses a `DBConnection` class to model all database connections.

Three concrete implementations of the `DataSource` interface are provided by the dbMapper package:

- `BasicDataSource`
- `ConnectionPoolDataSource`
- `JNDIDataSource`

`BasicDataSource` provides a single connection, while `ConnectionPoolDataSource` provides a pool of connections (which is useful for multi-threaded applications). `JNDIDataSource` is a mere wrapper around an installed `javax.sql.DataSource` that is bound to a JNDI path. These three `DataSource` implementations should satisfy the requirement of most database applications in acquiring and releasing database connections, although in some cases it may be desirable to create a custom data source.

Note that the flexibility of data sources allows an application to fine tune how the database resources (i.e. connections) are used. For example, in an area of the code where high performance is critical, an application might use a mapper whose context uses a dedicated `ConnectionPoolDataSource` instance, while all other areas of the code use mappers that share some other common data source. Alternatively, if there are other special processing needs regarding the management of database connections, a custom class that implements the `DataSource` interface could be implemented and used with the dbMapper package.

3.2.5 Mapping Set Files

A second main component of a mapping context is the mapping set. The mapping set, which consists of a set of mappings, provides all of the information needed by a mapper to persist objects for a specific set of Java classes. Each mapping defines the information needed to persist instances of a particular class. A mapping includes information such as which tables and columns of the database are to be used to store the various fields of the class.

The mapping set is specified in the mapper configuration file as a set of mapping set files. A *mapping set file* is an XML file that defines mappings for one or more classes. Note that although a complete mapping set may be specified in a single file, the dbMapper package allows a set of mapping set files to be used. Furthermore, within any mapping set file, any subset of mappings may be included or excluded from the set using include and exclude tags. Together, these options provide flexibility that lets the application programmer organize mappings in a way that best suits the application. For example, this makes it possible for two different mapping contexts to contain a common subset of mappings (i.e. two mapping sets could include the same mapping set file, so that both mapping sets contained that common subset of mappings).

3.2.6 Custom Database Processing: DAOs (Data Access Objects) and the `DAOInterface`

Although the `DefaultMapper` class provides a default implementation for all of the database operations defined by the DBInterface, there may be cases where an application programmer needs to provide his own implementation. For example, he may want to implement his own algorithm for persisting instances of some class that is too complex for the dbMapper object-relational mapping model. Or, for example, because of some other application-specific requirement, he may want to override the default processing

provided for a specific method of some specific class. The use of an interface named `DAOInterface` gives the programmer the capability to provide these custom implementations within the framework of the dbMapper package.

The `DAOInterface` interface is essentially identical to the `DBInterface` interface. (Refer to the javadoc API in the "doc/javadoc" directory for the specific definition of this interface.) An implementation of this interface is referred to as a *DAO*. The dbMapper package provides a single DAO, which is named `DefaultDAOImpl`. By default, each database operation that is invoked on a mapper uses a `DefaultDAOImpl` object to perform that operation. (It is the `DefaultDAOImpl` class that encapsulates all of the object-relational mapping processing that is done by the dbMapper package. While the transaction and data-source logic is still handled by the `DefaultMapper` class, the `DAOInterface` interface defines all of the methods needed to create, delete, update and query object instances of a given class.)

To override the default database operations provided by the dbMapper package, an application programmer must provide a custom DAO (i.e. a custom implementation of the `DAOInterface)`. To illustrate the point, we will use an example where an application programmer wants to override the `create`() method processing for a class named `MyClass`. As a first step, the application programmer must derive a new class from `DefaultDAOImpl`; we will call it `MyDAO`. Then he must override the `create`() method to implement his custom processing. (Of course, if a programmer would like to provide custom implementations for all of the methods, then he might prefer to directly provide an implementation of `DAOInterface`, rather than deriving a class from `DefaultDAOImpl`.) Finally, the programmer must modify the mapping for `MyClass` so that it uses `MyDAO` as its DAO (instead of the default DAO, `DefaultDAOImpl`). Refer to the "mappings and mapping Elements" section for details on how to specify a DAO for a class mapping.

As mentioned above, the `DAOInterface` method signatures are nearly identical to the `DBInterface` method. The only difference between the `DAOInterface` and DBInterface methods is that the `DAOInterface` methods have an extra argument of type `DBConnection`. To understand why, recall that all `DefaultMapper` operations occur within the context of a transaction. These transactions may be initiated by application code or by the `DefaultMapper` class itself to ensure the atomicity of `DBInterface` operations. The connection argument provides the application programmer of a DAO with the `DBConnection` object that is associated with the current database transaction (i.e. the one associated with the current thread of execution). The application programmer is expected to use this database connection so that the integrity of the transaction is maintained. (The programmer may create and use his own connection, but must realize that any operations done on that connection will not be part of the transaction being managed by the mapper that is calling the current DAO operation).

Also, an application programmer who implements a DAO is expected to throw exceptions in the DAO code to the calling `DefaultMapper` class. Otherwise, the integrity of the transaction maintained by the DefaultMapper class is not guaranteed (i.e. the commit or rollback might not give the desired result).

3.2.7 Configuring Mappers Via The Programming API

The previous sections introduced the mapper configuration file as a means to define mapping contexts. These contexts can then be used to instantiate individual mappers using the `createDefaultMapper` method of the `DBModule` class. Note, however, that the dbMapper package does provide a set of classes that let the application programmer specify mapping contexts and all associated data programmatically, instead of through configuration files. A key class for this type of development is the `ORMappingInfo` class. For more information regarding this class and related classes, refer to the Javadoc API documentation of the dbMapper package.

3.3 Creating a Mapper Configuration File

This section describes the content and format of mapper configuration files. For an overview of these files and how they are used, refer to the "Mapper Configuration Files" section. The DTD file for mapper configuration files is named "dbmapper.dtd", and is included in the "lib/dbmapper.jar" file of the dbMapper distribution.

A mapper configuration file is written in XML and composed of the following three elements:

- `logging` (optional)
- `data_sources`
- `mapping_contexts`

The `logging` element holds the `log4j` category name that dbMapper classes use to insert log and trace output into log files.

```
<!ELEMENT root (logging?,data_sources,mapping_contexts)>
```

The `data_sources` element consists of a set of `data_source` elements. Each `data_source` element contains an `id` attribute that uniquely identifies the `data_source`. Each `data_source` element defines a data source and can be any of the following types: `basic_data_source, connection_pool, jndi_data_source,` or `custom_data_source`.

The `basic_data_source` element represents a `BasicDataSource` object. The `basic_data_source` element is composed of a `connection_info_file` and a `max_connections` attributes. The `connection_info_file` attribute refers to a "database connection file"(refer to the "Creating a Database Connection File" section). The `max_connections` attribute specifies the maximum number of database connections that can be opened by the `BasicDataSource.`

The `connection_pool` element represents a `ConnectionPoolDataSource` object that manages a pool of database connections. The `connection_pool` element is composed of a `connection_info_file` and several pool capacity related attributes, namely `initial_capacity`, `capacity_increment` and `max_capacity`.

The `jndi_data_source` element describes a `JNDIDataSource` object. Each `jndi_data_source` element contains a single `jndi_location` attribute that is set to the JNDI-path of the installed `javax.sql.DataSource` object to which the `JNDIDataSource` is bound.

The `custom_data_source` element is used to define a custom or third-party `DataSource` implementation. The custom_`data_source` element is composed of a `class` attribute and `property` elements. The `class` attribute specifies the fully qualified class name of the `DataSource` interface implementation. Each `property` element represents a name/value pair setting that is used to customize the `DataSource` implementation. (Refer to the "custom_data_source Element" section for details.)

The `mapping_contexts` element consists of a set of `mapping_context` elements. Each `mapping_context` element defines a mapping context that may be used by the application to instantiate a mapper. Each `mapping_context` element is composed of an `id` attribute, a `data_source_id` attribute, and one `or_mapping_files` element. The `id` attribute uniquely identifies the mapping context among others. The `data_source_id` attribute contains a reference to a `data_source` element by specifying a data source `id` that is defined within the mapper configuration file. The `or_mapping_files` element represents the mapping set that is used by the mapping context. The `or_mapping_files` element is composed of `or_mapping_file` elements. Each

`or_mapping_file` element contains a reference to an OR mapping file (refer to the "Creating an Object-relational (OR) Mapping File" section for details). The `or_mapping_file` element may optionally contain an `includes_mapping_set` or `excludes_mapping_set` tag. These tags are used to select a specific subset of mappings from the file.

3.3.1 `logging` Element

Logging and tracing at run time are achieved by using the `log4j` package, which is a popular and widely used logging package for Java. (Please refer to `log4j` documentation at http://jakarta.apache.org/log4j/ for details on log category and other logging concepts. Some of the explanation below assumes familiarity with the `log4j` concepts). By default, all the dbMapper classes use the "`dbMapper`" category name for logging. This is done to provide control over the trace messages generated by the dbMapper classes at run time.

```
<!ELEMENT logging EMPTY>
<!ATTLIST logging category CDATA 'dbMapper'>
```

The `logging` element has a single attribute namely `category`.

Attribute	Description	Required
`category`	The `log4j` category name used by dbMapper classes to insert logging code.	No

3.3.2 `data_sources` and `data_source` Elements

The `data_sources` element contains all of the data sources defined for an application. It can contain any number of `data_source` elements. Each `data_source` element has a unique `id` attribute that can be used to reference it.

```
<!ELEMENT data_sources (data_source)+>
```

A `data_source` element encompasses all the necessary information needed to create an instance of the `DataSource` interface.

```
<!ELEMENT data_source
(basic_data_source|connection_pool|jndi_data_source|custom_data_source)>
<!ATTLIST data_source id CDATA #REQUIRED>
```

A `data_source` element has a single mandatory `id` attribute. The `id` attribute must contain a value that is unique among all data sources defined within the mapper configuration file.

Attribute	Description	Required
`id`	The `data_source` (unique) identifier.	Yes

A `data_source` element contains any of the following elements depending upon the data source type:

- `basic_data_source`
- `connection_pool`
- `jndi_data_source`
- `custom_data_source`.

The first three types correspond to the three concrete implementations of the `DataSource` interface provided by the dbMapper package. These three `DataSource` implementations should meet the

connection management requirements of most typical database applications. However, in some cases it may be desirable to create a custom class that implements the `DataSource` interface and associate it with the desired mapping contexts. The `custom_data_source` type encapsulates all the necessary information to create a custom `DataSource` implementation.

3.3.2.1 `basic_data_source` Element and `BasicDataSource`

The `BasicDataSource`, as indicated by its name, is a very basic implementation of the `DataSource` interface. The `BasicDataSource` simply establishes a new JDBC connection each time the `getConnection` method is invoked. The `releaseConnection` method simply frees all the resources acquired by the JDBC connection.

As opening a new JDBC connection is a costly operation, this data source is only suitable for those mappers (i.e. `DefaultMapper` objects) where database operations are infrequent or for one-time use only.

The `BasicDataSource` object requires that a database connection file be specified. That file supplies information that is used to locate and connect to a database, such as a JDBC driver name, URL, a user name and password, etc. See the "Creating a Database Connection File" section for details on the file format, parameters, and validation rules.

The `basic_data_source` element contains all of the information that is needed to create a `BasicDataSource` object. The following DTD snippet shows the attributes for this type of data source.

```
<!ELEMENT basic_data_source EMPTY>
<!ATTLIST basic_data_source connection_info_file CDATA #REQUIRED>
<!ATTLIST basic_data_source max_connections CDATA '0'> <!-- unlimited -->
```

The `basic_data_source` element has two attributes:

Attribute	Description	Required	Default
`connection_info_file`	The name of a database connection file.	Yes	N/A
`max_connections`	Maximum number of database connections that can be opened by this data source at any given time. A value less or equal to zero specifies unlimited connections.	No	Unlimited

3.3.2.2 `connection_pool` Element and `ConnectionPoolDataSource`

The `ConnectionPoolDataSource` class manages a pool of database connections so that database resources are efficiently managed. Pooling also allows concurrent database operations in multi-threaded applications.

A `ConnectionPoolDataSource` object creates a number of connections at startup (as specified by the `initialCapacity` parameter) and places them in a pool. When a user requests a connection, a free connection from the pool is returned to the user. When the user is done with a connection, he returns it back to the pool. This data source never closes connections, but does allocate and open new connections as required. This eliminates the overhead of closing and re-creating new connections for each request.

The `maxCapacity` parameter defines an upper bound on number of connections that may be created by this data source. A value less than or equal to zero is used to specify unlimited connections.

If all the pooled connections for a data source are in use at the time that another connection is requested, a `ConnectionPoolDataSource` object attempts to establish more database connections based on the value of the `capacityIncrement` property. However, the total number of connections can never exceed the `maxCapacity` value.

Here are some useful `ConnectionPoolDataSource` settings:

1. For a fixed size pool of `n` connections (all connections created at initialization time):
   `InitialCapacity = n; capacityIncrement = 0; maxCapacity >= n,`

2. For a growing pool with upper bound, `n`:
   `InitialCapacity = c; capacityIncrement >= 1; maxCapacity = n`
   `where 0 <= c <= n`

3. For an infinitely growing pool:
   `InitialCapacity >= 0; capacityIncrement >= 1; maxCapacity = 0`

Similar to the `BasicDataSource` data source, the `ConnectionPoolDataSource` also requires that a database connection file be specified. That file supplies information that is necessary to locate and connect to a database, such as its JDBC driver name, URL, a user name and password, etc. See the "Creating a Database Connection File" section for details on the file format, parameters, and validation rules.

The `connection_pool` element contains all the information needed to instantiate and configure a `ConnectionPoolDataSource` object.

```
<!ELEMENT connection_pool EMPTY>
<!ATTLIST connection_pool connection_info_file CDATA #REQUIRED>
<!ATTLIST connection_pool initial_capacity CDATA '1'>
<!ATTLIST connection_pool capacity_increment CDATA '1'>
<!ATTLIST connection_pool max_capacity CDATA '0'> <!-- unlimited -->
```

The `connection_pool` element has four attributes:

Attribute	Description	Required	Default
`connection_info_file`	The name of a database connection file.	Yes	N/A
`initial_capacity`	The initial capacity of the pool.	No	1
`capacity_increment`	The amount by which the capacity is increased when the more connections are needed.	No	1
`max_capacity`	The maximum number of database connections that can be opened by this data source at any given time. A value less than or equal to zero specifies unlimited connections.	No	Unlimited

3.3.2.3 `jndi_data_source` Element and `JNDIDataSource`

The `JNDIDataSource` data source adapts the `javax.sql.DataSource` interface to the dbMapper `DataSource` interface. The `JNDIDataSource` class constructor takes the JNDI path (i.e. location) of an installed `javax.sql.DataSource` as input. First, the constructor creates the initial naming context (`javax.naming.InitialContext`) from the `jndi.properties` file located in the application's run directory. Next, the `JNDIDataSource` constructor locates the installed

`javax.sql.DataSource` object that is bound to the JNDI path. The reference to the `javax.sql.DataSource` object is saved and kept for later reference.

The `JNDIDataSource` simply invokes the `getConnection()` method of the underlying `javax.sql.DataSource` object each time the `getConnection` method is invoked. The `releaseConnection` method simply frees all the resources acquired by the database connection.

The `jndi_data_source` element contains all the information that is needed to create a `JNDIDataSource` object.

```
<!ELEMENT jndi_data_source EMPTY>
<!ATTLIST jndi_data_source jndi_location CDATA #REQUIRED>
```

The `jndi_data_source` element has single `jndi_location` attribute:

Attribute	Description	Required
`jndi_location`	The JNDI-path of the installed `javax.sql.DataSource` object.	Yes

3.3.2.4 `custom_data_source` Element

In some cases, an application programmer may wish to create a mapper that uses a custom implementation of the `DataSource` interface. This may be done if none of the above (three) `DataSource` implementations provided by dbMapper meets the application's special processing needs.

The `custom_data_source` element is used to create and initialize such custom or third-party `DataSource` implementations. The dbMapper requires the user to specify the fully qualified name of the custom class that implements the `DataSource` interface. The custom implementation may define a set of simple name-value (`String`) properties to customize the `DataSource` object at instantiation. In order to pass these properties during object construction, the dbMapper expects the custom implementation to define a `public` constructor with following signature:

```
public <class name> (java.util.Properties)
```

```
<!ELEMENT custom_data_source (property)+>
<!ATTLIST custom_data_source class CDATA #REQUIRED>

<!ELEMENT property EMPTY>
<!ATTLIST property name CDATA #REQUIRED>
<!ATTLIST property value CDATA #REQUIRED>
```

The `custom_data_source` element has single mandatory `class` attribute:

Attribute	Description	Required
`class`	The fully qualified class name of the class that implements the `DataSource` interface.	Yes

The `custom_data_source` element can contain any number of `property` elements. These name/value pair properties are used to customize the `DataSource` during object instantiation. Each `property` element has two attributes:

Attribute	Description	Required
`name`	The name of the property to set.	Yes

`Value`	The value of the property.	Yes

3.3.3 `mapping_contexts` and `mapping_context` Elements

The `mapping_contexts` element lists a set of mapping contexts. This element may contain any number of `mapping_context` elements. Each `mapping_context` element defines a single mapping context. Each `mapping_context` element requires a unique `id` attribute value so that the element may be uniquely identified among other `mapping_context` elements defined in the same mapper configuration file. An application instantiates mappers that use specific mapping contexts by supplying the `id` of the desired mapping context in the `DBModule.createDefaultMapper` method.

```
<!ELEMENT mapping_contexts (mapping_context)+>
```

Each `mapping_context` element represents a mapping context that encapsulates the information needed by a mapper object to implement relation database persistence for a set of Java classes.

```
<!ELEMENT mapping_context (or_mapping_files)>
<!ATTLIST mapping_context id CDATA #REQUIRED>
<!ATTLIST mapping_context data_source_id CDATA #REQUIRED>
```

The `mapping_context` element defines two mandatory attributes, namely `id` and `data_source_id`. The `id` attribute uniquely identifies the `mapping_context` among others defined in the same mapper configuration file. The `data_source_id` refers to a `data_source element` that must be defined in the data source section of the mapper configuration file.

Attribute	Description	Required
`id`	The `mapping_context` identifier. It must be unique.	Yes
`data_source_id`	Identifier of the data source associated with this `mapping_context`.	Yes

The `mapping_context` element contains an element named `or_mapping_files`. That is described in the following section.

3.3.3.1 `or_mapping_files` and `or_mapping_file` Elements

An `or_mapping_files` element, which represents a mapping set, is defined by a set of `or_mapping_file` elements. Each `or_mapping_file` element specifies the name of an object-relational (OR) mapping file. (Refer to the "Creating an Object-relational (OR) Mapping File" section for details of OR mapping files).

By default, all class mappings of a mapping file specified by an `or_mapping_file` element are used. However, a specific subset of the class mappings may be selected by using either an `includes_mapping_set` or an `excludes_mapping_set` element. An `includes_mapping_set` element specifies a specific subset of mappings to be used from the file; all others are excluded. An `excludes_mapping_set` element is used to exclude specific mappings from the mapping set. Each mapping in an `includes_mapping_set` or an `excludes_mapping_set` element is specified by its `mapping class` and `mapping tag` attributes, which uniquely identify it.

```
<!ELEMENT or_mapping_files (or_mapping_file)+>
<!ATTLIST or_mapping_file path CDATA #REQUIRED>

<!ELEMENT includes_mapping_set (mapping)+>
```

```
<!ELEMENT excludes_mapping_set (mapping)+>

<!ELEMENT mapping EMPTY>
<!ATTLIST mapping class CDATA #REQUIRED>
<!ATTLIST mapping tag CDATA #IMPLIED>
```

Note that one motivation for defining a mapping set (i.e. an `or_mapping_files` element) as a set of files, is to allow different mapping sets to share commons subsets of mappings. This may be done by putting the mappings to be shared into one or more mapping files, then listing those same files in the different mapping set definitions.

Finally, note that every mapping set specified by an `or_mapping_files` element must satisfy the following two requirements. Otherwise, a run time exception will be thrown by the dbmapper package.

1. **A mapping set must not contain multiple mappings for the same Java class**. For example, if an `or_mapping_files` element specified two different mapping files, each containing a mapping for the same Java class, the resulting `or_mapping_files` element would not be valid. As another example, even if an `or_mapping_files` element contained a single file that defined two mappings for the same class (each with a different tag), that would also be an invalid.
2. **A mapping set must not contain unresolved mapping class references**. For example, an `or_mapping_files` element might contain a file that defines a mapping for a complex class, named `SomeClassA`, which contains a field of a class `SomeClassB`. If none of the mapping files specified in the `or_mapping_files` element contains a mapping for `SomeClassB`, then that `or_mapping_files` element is invalid.

3.4 Creating a Database Connection File

This section discusses the format, settings, and validation rules of the XML files used by the `BasicDataSource` and `ConnectionPoolDataSource` objects to locate and connect to a JDBC-capable relational database. These XML files, which are referred to as *database connection files*, specify the information necessary to obtain a connection to a database server. A database connection file specifies the following settings:

Setting	Description	Required
`Name`	The name of the database connection information.	No
`engine`	The persistence engine for the database server. At present, this setting is not used.	No
`driver`	The JDBC-driver class name for this data source. The driver is obtained from the JDBC `DriverManager` and must be located in the class path.	Yes
`url`	The JDBC URL for this data source of the form `jdbc:subprotocol:subname`.	Yes
`user_name`	The username used to log in to the database.	Yes
`password`	The password used to log in to the database.	Yes

3.4.1 DTD for Database Connection Files

For validation, database connection files should include the `"db_connection.dtd"` document type definition (DTD) provided with the dbMapper package. That file is included in the "lib/dbmapper.jar" file of the dbMapper distribution. The contents of the database connection file DTD is:

```
<?xml encoding="UTF-8"?>

<!ELEMENT connection_info
((driver,url,user_name,password)|(url,driver,user_name,password))>
<!ATTLIST connection_info name CDATA #IMPLIED >
<!ATTLIST connection_info engine CDATA #IMPLIED >

<!ELEMENT driver (#PCDATA)>

<!ELEMENT url (#PCDATA)>

<!ELEMENT user_name (#PCDATA)>

<!ELEMENT password (#PCDATA)>
```

3.4.2 Sample Database Connection File

For example, the following file could be used to obtain database connections using an Oracle 8 thin driver, use:

```
<?xml version="1.0"?>
<!DOCTYPE connection_info PUBLIC "DBMapper Database Connection"
"http://www.onsd.nec.com/software/db_connection.dtd">

<connection_info name="default" engine="oracle">
     <driver> oracle.jdbc.driver.OracleDriver </driver>
     <url> jdbc:oracle:thin:@myhost:1521:oracle_sid</url>
     <user_name> scott </userName>
     <password> tiger </password>
</connection_info>
```

3.5 Creating an Object-relational (OR) Mapping File

An *object-relational (OR) mapping file*, or simply *mapping file*, is an XML file that specifies the object-relational mappings for one or more Java classes. This section discusses the file format, settings, and validation rules of such files. To see how these files are used in mapper configuration files, refer to the "or_mapping_files and or_mapping_file Elements" section. The DTD for mapping files, which is named "db_or_mapping.dtd", is included in the "lib/dbmapper.jar" file of the dbMapper distribution.

An OR mapping file contains a single `mappings` element. The `mappings` element consists of a set of `mapping` elements. Each `mapping` element represents a mapping between a Java class and the relational table that will be used to store object instances of the class. Each `mapping` element is composed of a `class` attribute, a `table` attribute, an optional `pk_class` attribute, an optional `tag` attribute, an optional `dao_class` attribute and several `field` elements. The `class` attribute specifies the fully qualified class name of the class that is being mapped, which is sometimes referred to as the *mapped class*. The `table` attribute contains the name of the relational table that will be used to store object instances of the mapped class. The `pk_class` attribute specifies the primary key class, if any, for the mapping. The `tag` attribute is used to differentiate between two or more mappings defined for the same class. The `dao_class` attribute specifies the fully qualified class name of the DAO class tobe used for this mapping. Each `field` element represents a Java field of the mapped class and holds the information used to store it in the database. Each `field` element is composed of an `id` attribute, an `is_key` attribute, an optional `get_method` element, and an optional `set_method` element. The `id` attribute denotes the ID of the field being mapped. The value of `id` can be any string, but must be unique among the other field `id`

values for the class being mapped. The `is_key` attribute indicates whether the field is a key field. The `get_method` and `set_method` elements specify the field's accessor and modifier method names, respectively (as they appear in the Java code for the mapped class). Additionally, each `field` element must contains exactly one of the following elements, which specify the field type: `basic_type`, `nested_type`, `complex_type`, or `complex_collection_type`. (Follow the hyperlinks for a detailed description of these elements.)

3.5.1 `mappings` and `mapping` Elements

The `mappings` element is the root element of an OR mapping file. It can contain any number of `mapping` elements.

```
<!ELEMENT mappings (mapping)+>
```

The `mapping` element represents a class mapping and contains all of the information, such as table name, primary key, mapping tag, Java field mapping, etc., needed to map a Java object to a relational database. The class that is being mapped is referred to as a *mapped class*.

The dbMapper package does not create any of the relational tables that are specified in the mapping file. It is up to the application programmer to make sure that all of the tables specified in the mapping file are created with the appropriate key relations, indices, and database constraints before the mapping is actually used for any database operation. This approach was taken to provide maximum flexibility in creating database schema, and to decouple any database vendor specific dependencies (e.g. schema syntax, restrictions such as reserved keywords, table name length etc.) from the dbMapper package.

```
<!ELEMENT mapping (field)+>
<!ATTLIST mapping class CDATA #REQUIRED>
<!ATTLIST mapping table CDATA #REQUIRED>
<!ATTLIST mapping pk_class CDATA #IMPLIED>
<!ATTLIST mapping tag CDATA #IMPLIED>
<!ATTLIST mapping dao_class CDATA #IMPLIED>
```

A `mapping` element has five attributes:

Attribute	Description	Required
`class`	The fully qualified class name of the class that is being mapped.	Yes
`table`	The relational table that will be used to store object instances of the mapped class.	Yes
`pk_class`	The fully qualified class name of the primary key, if any, for the mapped class. For more information on primary key class, please refer to the "Primary Key" section.	No
`tag`	The mapping tag. This attribute is required when multiple mappings are defined for the same class. The combination of `class` and `tag`, which serves as an identifier for a mapping, must be unique among all mappings used by any single mapping context.	No
`dao_class`	The fully qualified class name of the DAO class for this mapping.	No

Note that the `dao_class` attribute is optional. If a value is not specified, an instance of the `DefaultDAOImpl` class is automatically instantiated and used. Otherwise, the `DefaultMapper` instance that uses this mapping will instantiate and use an instance of the specified DAO class. The specified DAO class must provide one of the following constructors. If both constructors are defined, the first will be used.

- **`public MyComplexDAO(DefaultMapper mapper).`** A public constructor with a single argument of type `DefaultMapper`. The `mapper` argument will contain a reference to the mapper that is instantiating this DAO instance.
- **`public MyComplexDAO()`**. A public constructor with no arguments.

The `mapping` element contains several `field` elements that map the Java fields of the mapped class to a relational table (column, row or set of rows). Only those Java fields that are intended to be stored in the database should be specified (i.e. any fields of a Java class that are not specified in the mapping are not persisted by the dbMapper package).

3.5.2 `field` Element

The `field` element specifies the mapping between a Java field and the relational table that will store the field value. Depending on the field type, a field may be stored in a single SQL column, a single table row, or a set of table rows.

```
<!ELEMENT field (get_method?,set_method?,
(basic_type|nested_type|collection_type|complex_collection_type))>
<!ATTLIST field id CDATA #REQUIRED>
<!ATTLIST field is_key (true|false) 'false'>
```

A `field` element has the following properties:

Attribute/ Element	Description	Required
`id`	Specifies the ID of the field that is being mapped. The value of `id` can be any string, unique among the other field `id` values for the class being mapped	Yes
`is_key`	Specifies whether this is a key field for the mapped class. A key field must be of basic type. By default, it is set to `'false'`.	No
`get_method`	Specifies an accessor method on the mapped class to be used to get the value of this field.	No
`set_method`	Specifies a modifier method on the mapped class to be used to set the value of this field.	No

When a `get_method` or `set_method` is not specified, the dbMapper package automatically constructs the names of those accessor and modifier methods (if and when they are needed) using the Sun JavaBean design pattern (i.e. the pattern where "`get`" or "`set`" is prepended to the field ID, with its first letter capitalized). For example, if a `get_method` is not specified for a field with an ID "`size`" of type `MyType`, the dbMapper automatically constructs and uses the following accessor method signature: "`public MyType getSize()`". Similarly, the automatically constructed modifier signature would be "`public void setSize(MyType)`". Thus, the `get_method` and `set_method` elements need only be specified when the mapped class does not provide these methods using the Sun JavaBean design pattern.

3.5.2.1 `is_key` Attribute

Some (but not all) of the `DBInterface` operations require the dbMapper to locate a unique database record that corresponds to the object being operated on. For example, if the `update()` method is called on some object, the dbMapper must be able to unambiguously identify the database record that needs to be updated. For this reason, the dbMapper uses the concept of primary keys and key fields as described in the

"Key Fields and Primary Keys" section. If the field represented by `field` is a key field, then its `is_key` attribute should be set to `true`. (If a mapping does not have any key fields, then that mapping does not specify a primary key.) Typically, it makes sense to define the primary key of a mapping to match the primary key used by the corresponding database table where the objects of the mapped class are stored.

As noted above, only some of the `DBInterface` operations require that a mapping include a primary key. Therefore, the specification of a primary key for a mapping is optional. Note however that for those mappings that do not specify a primary key, the following methods and features are not supported[1]:

- `update()`
- `updateTree()`
- `deleteByPrimaryKey()`
- `findAllPrimaryKeys()`
- `findByPrimaryKey()`
- `findPrimaryKeysByAttributes()`
- `findPrimaryKeysByQuery()`
- `delete()`
- `complex fields`
- `complex collection fields`

Note 1: If a primary key has more than one key field, then it is referred to as a composite primary key. Note that in this case, the user must define a primary key class to use those methods of the `DBInterface` that include the string "PrimaryKey" in their name. Refer to the "Key Fields and Primary Keys" section for a definition of "primary key class".

3.5.2.2 `get_method` Element

The `get_method` element specifies the accessor method of the mapped class for this `field` element.

```
<!ELEMENT get_method (#PCDATA)>
```

3.5.2.3 `set_method` Element

The `set_method` element specifies the modifier method name of the mapped class for this `field` element.

```
<!ELEMENT set_method (#PCDATA)>
```

3.5.2.4 Field Type

The dbMapper package provides persistence for four field types: basic, nested, complex, and complex collection fields. (Refer to the "Field Types" section for details.) This section defines the XML elements that specify the information needed to support persistence of these field types.

3.5.2.4.1 `basic_type` Element

The `basic_type` element contains all of the information needed to map a basic field to the database. The only information needed by the `DefaultMapper` class to map this type of field is the name of the database column used to store it. By default, the `DefaultMapper` class assumes that the name of the column is the same as the ID of the `field`. This default column name can be overridden by specifying the name of the database column in the `column` attribute. This information is captured in the DTD in the following lines:

```
<!ELEMENT basic_type (#PCDATA)>
```

```
<!ATTLIST  basic_type column CDATA #IMPLIED>
```

3.5.2.4.2 `nested_type` Element

A field is said to be a nested field if it is mapped to a single SQL column of the same database table that stores the containing object, and the field is nested within another field of the mapped class. Let us have another look at an example that was used in the "Field Types" section, to introduce some terminology associated with the `nested_type` element.

The following example uses the OR mapping between the `Person` class and the `person_table` table. The `Person` class contains five nested fields, namely `address.street, address.city, address.state, address.zip.zipCode` and `address.zip.zip4Code`:

```
class Zip {
       int zipCode;
       int zip4Code;
}
class Address {
       String street;
       String city,
       String state;
       Zip    zip;
}
class Person {
       String  firstName;
       String  lastName;
       int     socialSecNum;
       boolean isLiving;
       Address address;
}
create table person_table (
       firstname  VARCHAR(64),
       lastname   VARCHAR(64),
       ssn        INTEGER,
       living     CHAR(1),
       street     VARCHAR(64),
       city       VARCHAR(64),
       state      CHAR(2),
       zip        INTEGER,
       zip4       INTEGER
)
```

Notice that the `address.zip.zipCode` nested field (or simply `zipCode` nested field) is mapped to the `zip` SQL column of `person_table`. A nested field is described by listing all the intermediate fields separated by periods (i.e. dots), and finally the target field. This sequence of fields describing a nested field is called a *nested attribute path*. The intermediate fields nodes are referred to as *intermediate nodes* of the path, and the target field (last element in the path) is called the *leaf node*. For example, the `address.zip.zipCode` nested attribute path contains two intermediate nodes, namely `address` and `zip`, and a `zipCode` leaf node. The target field (i.e. the leaf node) in a path must be of basic type, so that it can be mapped to a single SQL column.

The `nested_type` element contains all of the information needed to map a nested field to the database.

```
<!ELEMENT nested_type ((intermediate_node)+,leaf_node)>
<!ATTLIST nested_type column CDATA #REQUIRED>
```

The `nested_type` element contains a single mandatory `column` attribute that specifies the database column to which the nested attribute is being mapped. The `nested_type` element consists of a series of `intermediate_node` elements, terminated by a `leaf_node`. These node elements capture the nested attribute path as described above.

```
<!ELEMENT intermediate_node (get_method?)>
<!ATTLIST intermediate_node node_id CDATA #REQUIRED>
<!ATTLIST intermediate_node class CDATA #REQUIRED>
```

An `intermediate_node` of a nested path has three settings, `node_id`, `class`, and `get_method`. The `class` attribute specifies the fully qualified class name of `intermediate_field`. The `id` attribute is the ID of `intermediate_node`. The optional `get_method` element specifies the accessor method that is used by the parent class to access the value of this node. If `get_method` is not specified, the dbMapper package assumes that the mapped class provides such a method, and that its signature follows the Sun JavaBean pattern. Refer to the "Field Element" section for a description of the Sun JavaBean pattern, and an example.

Setting	Description	Required
`node_id`	The ID of the intermediate node.	Yes
`class`	The fully qualified class name of the intermediate field/node.	Yes
`get_method`	The accessor method name for this intermediate node.	No

```
<!ELEMENT leaf_node (get_method?,set_method?)>
<!ATTLIST leaf_node node_id CDATA #REQUIRED>
<!-- leaf_node class should be of basic_type -->
<!ATTLIST leaf_node class CDATA #REQUIRED>
```

A `leaf_node` element contains all the `intermediate_node` settings plus an optional `set_method` setting. The `set_method` element specifies the modifier method that is used by the parent object to set the value of this `node`. If `set_method` is not specified, the dbMapper package assumes that the mapped class provides such a method, and that its signature follows the Sun JavaBean pattern. Refer to the "field Element" section for a description of the Sun JavaBean pattern, and an example.

Setting	Description	Required
`node_id`	The ID of the leaf node.	Yes
`Class`	The fully qualified class name of the leaf field/node. The class must be one that corresponds to a basic field type. Refer to the "Basic Field" section for a list of such types.	Yes
`get_method`	The accessor method name for this leaf node.	No
`set_method`	The modifier method name for this leaf node.	No

In our example, the dbMapper would simply use the following code to access the `address.zip.zipCode` nested field of a `Person` object (assuming that no custom get or set methods are specified for these fields):

```
person.getAddress().getZip().getZipCode()
```

If any of the intermediate accessor methods return a `null` object, the dbMapper acts as if the leaf field (`zipCode`) was `null`.

To modify the `address.zip.zipCode` nested field in a `Person` object, the dbMapper would simply use:

```
person.getAddress().getZip().setZipCode(newZip)
```

Note that the dbMapper package never attempts to instantiate intermediate nodes for a leaf node. For example, if the `DBInterface.create()` method is invoked on an object with nested fields whose intermediate nodes have not been instantiated, the leaf nodes will not be loaded from the database. In fact, any time that a get method for an intermediate node returns a `null` value, the attempt to reach the leaf node is terminated. Thus, it is up to the application programmer to ensure that intermediates nodes are instantiated, if any operations on leaf nodes are to be executed. Also, note that the dbMapper does not treat a `null` return value from a get method as an error. Rather, the processing for the leaf node is simply considered to be complete, even though the leaf node was never reached.

3.5.2.4.3 `complex_type` Element

The `complex_type` element contains the information needed to map a complex field to the database. A `complex_type` element consists of one `element_mapref` element and one `key_bindings` element. The `element_mapref` element specifies the mapping to be used for this complex field. In other words, the complex field class must be a mapped class itself. The mapping of this mapped class is used to persist the complex field value to the database. The `key_bindings` element specifies the relationship between the key fields of the containing class and the fields of the complex field itself.

```
<!ELEMENT complex_type (element_mapref,key_bindings)>
```

Please refer to the "element_mapref Element" and "key_bindings Element" sections for details.

3.5.2.4.4 `element_mapref` Element

When specifying the mapping for a complex or complex collection field, one must specify the exact mapping that should be used to persist the field. If only a single mapping exists for the Java class of the complex field, then it is sufficient to specify only the `class` attribute to uniquely specify that mapping. However, if more than one mapping exists, it is necessary to specify both the `class` and `tag` attributes to uniquely identify the mapping.

```
<!ELEMENT element_mapref EMPTY>
<!ATTLIST element_mapref class CDATA #REQUIRED>
<!ATTLIST element_mapref tag CDATA #IMPLIED>
```

The `element_mapref` element has two attributes:

Attribute	Description	Required
`class`	The fully qualified class name of the field that is being mapped.	Yes
`tag`	The mapping tag. This attribute is needed if the desired mapping uses a non-default tag.	No

3.5.2.4.5 `key_bindings` Element

The dbMapper assumes that complex fields are not stored in the same table as the containing object. Because of this assumption, the dbMapper needs enough information to unambiguously correlate complex field entries (which reside in one table) to their containing objects (which reside in a different table). The

dbMapper maintains this association by requiring that each complex field store the key of its containing object.

This association between the complex field and its containing class is referred to as a *key binding*, and is represented by the `key_bindings` element. The `key_bindings` element is composed of multiple `key_binding` elements. Each `key_binding` element associates one fieldsof the containing class, represented by the `parent_field` element, with a field of the complex field, represented by the `child_field` element. Normally, a `key_binding` element is specified for each of the key fields of the containing class. In other words, `N` `key_binding` elements would normally be defined for a parent class with a composite key consisting of `N` fields.

```
<!ELEMENT key_bindings (key_binding)+>

<!ELEMENT key_binding EMPTY>
<!ATTLIST key_binding parent_field CDATA #REQUIRED>
<!ATTLIST key_binding child_field CDATA #REQUIRED>
```

As an example, consider a `Person` class that contains a complex field, `house`, of the type `House`. Assume that the `Person` class has a basic field called, `name`, which is its key. Also, assume that the `House` class has a basic field (of the same type) called `owner`. The `owner` field is used to store the `name` value of the containing `Person` object. In this case, the dbMapper package must know that the `owner` field of the `House` class corresponds to the `name` field of the `Person` class. This type of information is captured in the `key_bindings` element. For this example, the `House` class would define a single `key_binding`, with `name` as the value of the `parent_field` element, and `owner` as the value of the `child_field` element.

3.5.2.4.6 `complex_collection_type` Element

The `complex_collection_type` element contains the information needed to map a complex collection field to a relational database.

```
<!ELEMENT complex_collection_type
((element_mapref,container_class,key_bindings)
|(container_class,element_mapref,key_bindings))>
```

The `complex_collection_type` element is composed of one `element_mapref` element, one `container_class` element, and one `key_bindings` element.

The `element_mapref` element specifies the OR mapping that is used to persists the elements of the array or collection to the database. Please refer to “element_mapref Element” section for details.
The `container_class` element specifies the collection type.

```
<!ELEMENT container_class (#PCDATA)>
```

If the fields are stored in an array, the dbMapper expects the following signature for `container_class`:

> <fully qualified class name of the array elements> [].

Note that the class name will be the same as that used for the `element_class` element, and that the value ends with "[]".

The following example shows how the container class for an array of `mypkg.MyClass` objects would be specified:

```
<container_class> mypkg.MyClass[] </ container_class>
```

When the complex collection field uses a collection type (instead of an array), the `container_class` must specify the fully qualified class name of the container class that is used. The dbMapper expects the container class to implement the `java.util.Collection` interface and to provide a public default constructor (i.e. a constructor with no arguments). The following example shows how the container class would be specified for a complex collection field that uses the `java.util.ArrayList` class as the container.

```
<container_class> java.util.ArrayList </container_class>
```

The `complex_collection_type` element defines a `key_bindings` element to specify the relationship between the entries stored in the collection/array and the key fields of the containing class. Please refer to the "key_bindings Element" section for details.

3.5.3 Primary Key Class

Although the dbMapper does not mandate that an OR mapping define a primary key class, in some cases it may be desirable. As the primary key represents the identity of a persistent user object, a client application may save this data elsewhere, e.g. in memory or in a file, so that the key may later be used to obtain the user object from the database. The primary key can also be useful in cases when a user object contains many attributes and requires a large amount of memory. In this case, it may make sense for the application to maintain a collection of keys, rather than a collection of the entire object instances. Specific object instances can then be retrieved, as needed, using the keys in the collection.

Note that a class mapping must be provided for any primary key class that is used by the dbMapper package.

4 Developing With dbMapper

To use dbMapper, you will need access to a relational database server and Java class libraries that include a JDBC 2.0 compliant driver class. The examples provided with the dbMapper distribution have been tested with a variety of JDBC-capable RDBMS products, such as Oracle 8.0 and 8i (http://www.oracle.com/), MySQL 3.23.39 (http://www.mysql.com/), PostGres 7.2.2 (http://www.postgresql.org/), HyperSonic Database 1.7.1(http://hsqldb.sourceforge.net/).

The dbMapper distribution includes all of the files needed to run the examples, including database schema files that may be used to create the relation tables required to run the examples. Those files are located in the `sql` directory. A sample database connection file is also provided for Oracle, MySQL, PostGres and HyperSonic database products. These file are used to locate and connect to database servers. You should edit one of these files, or create a new one to reflect the settings of the particular relational database server that you plan to use. The dbMapper distribution also includes the mapper configuration and OR mapping files used in the examples.

The database connection file used in the following examples uses the Oracle 8i thin driver class provided by Oracle. However, the examples may be run with any RDBMS product that comes with a JDBC 2.0 complaint drive, provided that the database connection file is modified accordingly, and the schema file is modified, if needed, to support any vendor-specific syntax. Before running any of the examples, first make sure the following system requirements are met:

4.1 System Requirements

To use dbMapper, you must add the following libraries to your `CLASSPATH`.

Jar Name	Needed For	Available At	Included with the distribution?
`src.jar`, `tools.jar`	JDK 1.3 or later	http://java.sun.com/	No
`dbmapper.jar`	The dbMapper library	with the distribution	Yes
`log4j.jar`	apache `log4j` classes (1.1.3 or later) used by `dbmapper.jar`	http://jakarta.apache.org/log4j/	Yes
`xerces.jar`	Xerces XML parser (1.4.3 or later)	http://xml.apache.org/xerces/	Yes
`jdom.jar`	jdom 1.0 or later, parse *OR* *mapping XML file*	http://jdom.org/	Yes
JDBC driver library provided by database vendor	database tasks	(Ex: Oracle drivers http://www.oracle.com)	No

The binary distribution of dbMapper includes all the jar files except the database vendor JDBC driver libraries. Refer to your database product manuals to locate the JDBC library (with `.jar` or `.zip` extension). For example the Oracle JDBC library, can be located at:

```
$ORACLE_HOME/jdbc/lib/classes12.zip (Unix) OR
%ORACLE_HOME%\jdbc\lib\classes12.zip (Windows).
```

where `ORACLE_HOME` is the directory where the Oracle server or client is installed.

4.2 Installation Tasks

The dbMapper distribution includes a number of examples that are easy to run, and demonstrate major features of dbMapper package. You can review the Java source files of the examples to see how they work.

Before running any of the examples, please finish the following tasks:

1. Set up your `classpath` (see "System Requirements" section). Make sure '`.`' (current directory) is in your `classpath`.
2. Make sure the JDK is in your `path`. Set the JAVA_HOME environment variable to the directory where the JDK is installed. Go to the JDK installation page (http://java.sun.com/j2se/1.3/install.html) and follow the directions for your platform.
3. Locate the JDBC library (with `.jar` or `.zip` extension) for your RDBMS vendor. Add this to your `classpath`.

Some additional tasks need to be completed before the examples can be run. Refer to the "Running the Examples" section for details.

4.3 Building the dbMapper Package

This section provides a reference to the build options that are provided with the dbMapper package. However, note that unless the Java source code files are modified, there is no need to run any build commands, as the dbMapper distribution itself includes compiled code and javadoc for all classes. However, you will need to compile the examples, if you wish to run them. Refer to the next section for instructions.

Before attempting any of the build commands, be sure to first finish the installation tasks of the previous section. When finished, go to the root directory of dbMapper distribution.

To build the dbMapper package you must have `Ant` 1.2 or later installed. You can download the latest version of `Ant` from http://jakarta.apache.org/builds/ant. Make sure that `Ant` is in your `path`. Also, set the `ANT_HOME` environment variable. This should be set to the directory where `Ant` is installed.

To build the entire dbMapper distribution (library jar files and javadoc API documentation), simply type
```
ant
```
To build only the dbMapper library jar files, use
```
ant lib
```
To build only the javadoc API documentation, use
```
ant javadoc
```
To remove all class files, including those in the original distribution, use
```
ant clean
```

4.4 dbMapper Examples

The dbMapper distribution includes a set of examples that demonstrate various capabilities of the dbMapper package. Each example is contained in a subdirectory of the "examples" directory. This section describes the organization of the examples, and how to run them. Note that each example provides both "bat" files and "sh" files, so that the examples can be run in Windows and Unix environments. This document assumes that the user is working in the Windows environment, and so will refer to the "bat" files only.

4.4.1 Running the Examples

This section describes how to run the examples. Before running them, you will first need to set up your environment properly, and compile them. The following subsections provide instructions to do this. Note that the last example, which is an EJB example, is organized slightly differently. Refer to the EJB example section for details.

4.4.1.1 Setting Up Your Environment

Before running the examples, it is necessary to set some environment variables. To do this, first open up the `myenv.bat` file located in the root directory of the dbMapper distribution. Examine the various settings, and adjust them as needed for your configuration. After saving your editing changes, run `myenv.bat`. This will give you the environment needed to run the examples.

4.4.1.2 Example Directory Structure

Each example has its own directory under the `examples` directory. The name of the example directory gives some indication of the principles that the example demonstrates. Each example directory contains the following subdirectories:

- **src** – the source code for the example.
- **classes** - the compiled class files for the source files in "src"
- **data** - the XML data files needed for the example

4.4.1.3 Compiling an Example

To compile an example, simply run the `compile.bat` script in the example directory. The compiled "class" files will be written to the `classes` directory.

4.4.1.4 Running an Example

Before running an example, you must take care of two more things. First, you must create the tables needed by the examples. To do this, use the `create.sql` schema file in the `sql` directory.

Secondly, you must modify the `data/db_connection.xml` file of the example to match your database settings. The files included with the distribution contain settings for an Oracle database. If you are using a different database product, you must modify `data/db_connection.xml` to use the settings for that database product. Examples for several database products are provided in the subdirectories of the `sql` directory. Also, if you plan to use the same database and database user for all of the examples, you may want to copy the modified connection file that you create to the `data` directories of the other examples, as well.

If your database is up and running, you can now run the example by executing the `run.bat` script. The output that is written to your terminal is also written to the `dbdemo.log` file.

4.4.2 Example1 – DataSources

In this example, we demonstrate how to create and configure the different types of data sources provided by the dbMapper package. If you have not done so already, please first read the "data_sources and data_source Element" section. Later in this example, we will also create and use a custom implementation of the `DataSource` interface.

Before using any of the dbMapper classes, the dbMapper package should first be initialized by invoking the `static init(String mapperConfigFile)` method of the `DBModule` singleton class. The `init` method takes a mapper configuration file as the only argument. The following code snippet is taken from the `Test.java` file (in the `examples\ex01-datasources\src` directory):

```
      DBModule dbm = DBModule.init (mapperFile); // mapperFile is set to
data/dbmapper.xml file path
```

Before proceeding any further, let us first look at the content of the `dbmapper.xml` file:

```
<?xml version="1.0"?>
<!DOCTYPE root PUBLIC "DBMapper Config" "http://www.onsd.nec.com/software/dbmapper.dtd">
<root>
```

The first line is simply an XML prolog or header statement that indicates that our document uses version 1.0 of XML. The second line in above snippet indicates that the XML document is validated using the "`dbmapper.dtd`" DTD file and "`root`" is the root element of our document.

```
<data_sources>

   <!-- basic connection manager with no maximum bound on connections -->
   <data_source id="ds_basic">
      <basic_data_source
         connection_info_file="data/db_connection.xml"
      />
   </data_source>
```

All the data sources managed by the singelton `DBModule` object are defined within the `<data_sources> </data_sources>` XML tags. The last five lines in the above snippet define a `BasicDataSource` object, named `ds_basic`, that has no upper bound on the number of connections held open by this data source at any given time.

```
   <!-- connection pool -->
   <data_source id="ds_pool">
      <connection_pool
              connection_info_file="data/db_connection.xml"
         initial_capacity="0"
      />
   </data_source>
```

The above creates an infinitely growing `ConnectionPoolDataSource` identified by `ds_pool`. The `ds_basic` data source provides a single database connection, while the `ds_pool` data source provides a pool of connections, which may be useful for a multi-threaded application.

```
   <!-- custom datasource : you can modify this entry according to your
                            DataSource settings (class, jdbc information..)
   -->
   <data_source id="ds_custom">
      <custom_data_source class="MyDataSource">
.................................
      </custom_data_source>
   </data_source>
```

The above defines a custom (user-defined) implementation of the `DataSource` interface, identified by `ds_custom`. The XML settings and `DataSource` implementation are discussed in detail at the end of this section.

```
</data_sources>

<mapping_contexts>

   <mapping_context id="custom" data_source_id="ds_custom">
      <or_mapping_files>
         <or_mapping_file path="data/or_mapping.xml"/>
      </or_mapping_files>
   </mapping_context>

   <mapping_context id="basic" data_source_id="ds_basic">
      <or_mapping_files>
         <or_mapping_file path="data/or_mapping.xml"/>
      </or_mapping_files>
   </mapping_context>

   <mapping_context id="pool" data_source_id="ds_pool">
      <or_mapping_files>
         <or_mapping_file path="data/or_mapping.xml"/>
      </or_mapping_files>
   </mapping_context>

</mapping_contexts>
```

The first line in the above document fragment marks the end of the data source declarations. All of the mapping contexts managed by the singelton `DBModule` object are defined within the `<mapping_contexts> </mapping_contexts>` tags. The next few lines declare three mapping contexts, namely `custom`, `basic` and `pool`. Each of these mapping contexts specify a data source (defined within the `<data_sources>..</data_sources>` tags) and a mapping set (as defined by a set of mapping files). For example, the "`pool`" mapping context is created by specifying the "`ds_pool`" data source and the mapping set specified by the single mapping file named "`or_mapping.xml`".

(Mapping sets and the mapping files that define them are discussed in detail in the next example(s). In this example we will only concentrate on the data sources.)

```
</root>
```

The above line marks the end of the mapper configuration file.

Now lets get back to the rest of the code in `Test.java`. The next few lines show how to create a mapper by specifying the mapping context identifiers defined in the mapper configuration file.

```
      DBInterface poolDBIf = dbm.createDefaultMapper ("pool");
      DBInterface basicDBIf = dbm.createDefaultMapper ("basic");
      DBInterface customDBIf = dbm.createDefaultMapper ("custom");
```

The following code fragment performs a very basic test on these three mappers. For each mapper, the code simply attempts to establish then release four database connections. This process is repeated one hundred times in a loop. Upon invocation of the `getConnection/releaseConnection` methods, the mapper simply returns the result of the `getConnection/releaseConnection` invocation on the underlying `DataSource` object.

```
      testDBIf (poolDBIf, "pool");
      testDBIf (basicDBIf, "basic");
      testDBIf (customDBIf, "custom");
   }
   void testDBIf (DBInterface dbIf, String ifName) throws Exception {
      DBConnection[] connections = new DBConnection[4];
      long startTime = System.currentTimeMillis();
      for (int i=0; i < 100; i++) {
         for (int j=0; j < connections.length; j++) {
            connections[j] = dbIf.getConnection();
         }
         for (int j=0; j < connections.length; j++) {
            dbIf.releaseConnection (connections[j]);
         }
      }
      Logger.debug ("ifName[" + ifName + "] time in ms = "
                  + (System.currentTimeMillis()-startTime));
   }
```

After executing this example program, one can verify from the messages in the output log that the database resources are most efficiently managed by the `ConnectionPoolDataSource` data source, which represents a pool of database connections.

Now let us look at the custom data source declared in the mapper configuration file:

```
   <data_source id="ds_custom">
      <custom_data_source class="MyDataSource">
         <property name="driver" value="oracle.jdbc.driver.OracleDriver"/>
         <property name="url"
value="jdbc:oracle:thin:@myhost.mydomain.com:1521:oracle_sid"/>
         <property name="user" value="scott"/>
         <property name="password" value="tiger"/>
      </custom_data_source>
   </data_source>
```

The dbMapper requires that the data source class, `MyDataSource`, implement all methods of the `DataSource` interface. The data source class is also expected to provide a public constructor that takes a single input argument of type `java.util.Properties`. When creating the custom data source object, dbMapper (i.e. the `DBModule` object) will use:

```
        java.util.Properties props = new java.util.Properties();
```

```
        props.setProperty (“driver”, “oracle.jdbc.driver.OracleDriver”);
        props.setProperty (“url”,
                           "jdbc:oracle:thin:@myhost.mydomain.com:1521:oracle_sid");
        props.setProperty (“user”, “scott”);
        props.setProperty (“password”, “tiger”);
        DataSource ds = new MyDataSource (props);
```

The content of the custom data source class `MyDataSource` follows:

```
...... // Import statements
public class MyDataSource implements DataSource {
   java.util.Properties connectionArgs = null;
   ...........
   public MyDataSource (java.util.Properties props) throws Exception {
      String jdbcDriverClassName = props.getProperty ("driver");
      connectionArgs = props;
      // Load the JDBC driver
      Class.forName (jdbcDriverClassName);
   }

  /*********** Implementation of DataSource methods ************/
   // Returns a new JDBC connection wrapped inside DBConnection.
   public DBConnection getConnection() {
      DBConnection dc = null;
      String jdbcURL = connectionArgs.getProperty ("url");
      try {
         java.sql.Connection conn = DriverManager.getConnection (jdbcURL,
                                                        connectionArgs);
         dc = new DBConnectionImpl (conn);
      }
      catch (Exception e) {
         Logger.error ("Error creating connection. Details: ", e);
      }
      return dc;
   }

   //Release an previously opened DBConnection.
   public void releaseConnection (DBConnection conn) {
      ((DBConnectionImpl) conn).destroy();
   }
   ......
}
```

4.4.3 Example2 – Basic Type

This example uses a very simple class named `Primitive.java,` which has two basic fields, to demonstrate step by step, the simplicity of using the dbMapper package. Lets first look at `Primitive.java`:

```
public class Primitive {
   int x; // Primary key attribute
   String y;

   // The no-argument constructor, required by dbMapper
   public Primitive() { x = 0; y = null; }

   public Primitive(int _x, String _y) { x = _x; y = _y; }
    .......
   // Accessor methods
   public int getX() { return x; }
   public String getY() { return y; }
   // Modifier methods
   public void setX(int _x) { x = _x; }
   public void setY(String _y) { y = _y; }
   .....
}
```

`Primitive` objects are persisted in the `demo_primitive` table, whose schema is defined by the "`create.sql`" file (located in the `samples` directory):

```
create table demo_primitive (
   col_x INTEGER PRIMARY KEY,
   col_y VARCHAR(64)
)
```

The `demo_primitive` table defines two columns, named `col_x` and `col_y`, to store the `x` and `y` fields respectively. A primary key (composed of single column, `col_x`) is created on the table to emphasize that `x` is a key field, i.e. a `Primitive` object can be located in database by specifying the value of the field `x`.

The mapping file (located in `examples/ex02-basic_fields/data` directory) specifies that the `demo_primitive` table is to be used to store instances of the `Primitive.java` class:

```
<?xml version="1.0"?>
<!DOCTYPE mappings PUBLIC "DBMapper OR Mapping"
"http://www.onsd.nec.com/software/db_or_mapping.dtd">
<!-- Set DTD validation file -->

<mappings>

<!-- Bind Primitive class to demo_primitive table -->
<mapping class = "Primitive"
       table="demo_primitive" >
      <!-- bind java int x to sql column col_x:int (indicate x as key
           attribute -->
      <field id="x" is_key="true" >
                <basic_type column="col_x">int</basic_type>
      </field>
      <!-- bind java String y to sql column col_y:varcahr2(64)-->
      <field id="y">
                <basic_type column="col_y">String</basic_type>
      </field>
</mapping>
</mappings>
```

Since the `Primitive` mapping has only one key field, `x`, the `java.lang.Integer` class (i.e. the wrapper class for the primitive `int` type) can be used as the primary key for this mapping (see the "Key Fields and Primary Key" section). Also note that the default column name for the field `x` is overridden by `col_x` using the `column` attribute, as described in the "basic type Element" section. (Similarly, the default column name for `y` is overridden by `col_y`.)

The following code snippet shows how `Primitive` object instances are created, deleted, modified, and located in the `demo_primitive` table. First, the `main` method instantiates a `Test` object by passing a mapper configuration file as the only argument. This file is used to initialize the `DBModule` class and to create a mapper object (as described in the previous example). Next, the `test()` method of the `Test` class is invoked to execute the example code.

```
public class Test {
   DBInterface mapper; // mapper used for this test

   Test (String mapperFile) {
      // create the mapper to be used for this test
      try {
         DBModule dbm = DBModule.init (mapperFile);
         mapper = dbm.createDefaultMapper ("default");
         Logger.debug (mapper.toString());
      }
      catch (Exception e) {
         Logger.error ("Error details: ", e);
         System.exit(0);
      }
```

```
   this.mapper = mapper;
}

public static void main (String[] args) {
   try {
      String mapperFile = args[0];
      Test testSuite = new Test(mapperFile);
      testSuite.test();
   }
   catch (Exception e) {
      Logger.error ("Error details: ", e);
   }
}

void test() throws Exception {
   AttrValMap aValMap = null;
   HashMap    hValMap = null;

   // Delete older entries
   mapper.deleteAll (Primitive.class);

   // Create a Primitive object and persist it to database
   Primitive o1 = new Primitive(3, "test");
   mapper.create (o1);
   Logger.debug ("create(Object) :: Object " + o1
                + " is successfully created.");

   // Delete the o1 Primitive object created above
   mapper.delete (o1);
   Logger.debug ("delete(Object) :: Object " + o1
                 + " is successfully deleted.");

   // Create o1 again and then delete it using primay key
   mapper.create (o1);
   mapper.deleteByPrimaryKey (new Integer(3), Primitive.class);
   Logger.debug ("delete(pk,cls) :: Object " + o1
               + " is successfully deleted.");

   // Create o1 again. Delete all the Primitive objects with x equal
   // to 3 (deletes o1 from database)
   mapper.create (o1);
   aValMap = new AttrValMap();
   aValMap.put ("x", 3);
   mapper.deleteByAttributes (aValMap, Primitive.class);
   Logger.debug ("deleteByAttributes(aval,cls) :: Object " + o1
                + " is successfully deleted.");

   // Create o1 again. Update o1 attributes and update new changes
   // to database
   mapper.create (o1);
   Logger.debug ("create(Object) :: Object " + o1
                + " is successfully created.");
   o1.setY ("New Value");
   mapper.update(o1);
   Logger.debug ("update(Object) :: Object " + o1
                + " is successfully updated.");

   // Update selected o1 attributes to database (using AttrValMap)
   aValMap = new AttrValMap();
   aValMap.put ("y", "AttrVal update");
   mapper.update(o1, aValMap, true);
   Logger.debug ("update(Object,aval,bool) :: Object " + o1
                + " is successfully updated.");

   // Update selected o1 attributes to database (using HashMap)
   hValMap = new HashMap(1);
   hValMap.put ("y", "HashMap update");
   mapper.update(o1, hValMap, true);
   Logger.debug ("update(Object,hval,bool) :: Object " + o1
                + " is successfully updated.");
```

```
// Create few more Primitive objects and persist them to database
Primitive o2 = new Primitive(6, null);
mapper.create (o2);
Primitive o3 = new Primitive(36, "Welcome !");
mapper.create (o3);
Primitive o4 = new Primitive(36*6, "Hi");
mapper.create (o4);

// Locate an previously created Primitive object in database using
// primary key
Primitive retVal = (Primitive) mapper.findByPrimaryKey (
            new Integer(36), Primitive.class); // should return o3
Logger.debug ("findByPrimaryKey(pk,class) :: Object " + retVal
             + " is successfully found.");

// Locate an previously created Primitive object in database by
// specifying key attribute x
aValMap = new AttrValMap();
aValMap.put ("x", 36*6);
Collection c1 = mapper.findByAttributes (aValMap, Primitive.class);
// c1 should contain only o4
Logger.debug ("findByAttributes(aval,class) :: Collection "
              + c1.toString() + " is successfully found.");

// Find all the Primitive object in database with null y (non-key
// attribute)
aValMap = new AttrValMap();
aValMap.put ("y", null);
Collection c2 = mapper.findByAttributes (aValMap, Primitive.class);
// c2 hould contain only o2
Logger.debug ("findByAttributes(aval,class) :: Collection "
              + c2.toString() + " is successfully found.");

// Find Primitive objects based on a custom SQL query
Collection c3 = mapper.findByQuery (
         "SELECT * from demo_primitive where col_x >= 36 order by col_x",
         Primitive.class);
Logger.debug ("findByQuery(query,class) :: Collection "
              + c3.toString() + " is successfully found.");

// Find all the Primitive objects stored in demo_primitive table
Collection c4 = mapper.findAll (Primitive.class);
Logger.debug ("findAll(class) :: Collection " + c4.toString()
             + " is successfully found.");

// Find a primary key object based on non-key attribute y.
// And then delete the corresponding object from database.
aValMap = new AttrValMap();
aValMap.put ("y", "Hi");
Collection c5 = mapper.findPrimaryKeysByAttributes (
                                        aValMap, Primitive.class);
// Above method should return o4. Remove it
Logger.debug ("findPrimaryKeysByAttributes(aval,class) :: Collection "
             + c5.toString() + " is successfully found.");
Object akey = c5.iterator().next();
mapper.deleteByPrimaryKey (akey, Primitive.class);

// Find primary key objects based on a custom SQL query
Collection c6 = mapper.findPrimaryKeysByQuery (
            "SELECT col_x from demo_primitive where col_y IS NOT NULL",
                 Primitive.class);
Logger.debug ("findPrimaryKeysByQuery(query,class) :: Collection "
            + c6.toString() + " is successfully found.");

// Find primary key of all the Primitive objects stored in
// demo_primitive table
Collection c7 = mapper.findAllPrimaryKeys (Primitive.class);
Logger.debug ("findAllPrimaryKeys(class) :: Collection "
            + c7.toString() + " is successfully found.");

// Add "36*6" back.
```

```
      mapper.create (o4);
      Collection c8 = mapper.findAll (Primitive.class);
      Logger.debug ("findAll(class) :: Collection " + c8.toString()
                  + " is successfully found.");
   }
}
```

4.4.4 Example3 – User Class (User-defined Primary Key Class and Basic Types)

This example uses a class named `User` to illustrate the use of a user-defined primary key class and the basic types supported by the dbMapper. The `User` class declares several basic fields, and instances of the `User` class are mapped to the `demo_user` database table. The `demo_user` table's primary key is a composite key of the `firstName`, `lastName`, and `pin` columns.

```
public class User {
   // Primary key attribute
   String firstName;
   String lastName;
   long pin;

   // other attributes
   char sex;
   double height;
   String email;
   boolean alive;
   Integer income;
   Short dob;

   // The no-argument constructor, required by dbMapper
   public User() { }
   // Other constructors
   ......

   // Getter/Setter methods for the attributes
   ......
}
```

The `User` class attributes are mapped to the `demo_user` table columns as shown in the following SQL statement:

```
create table demo_user (
   firstName VARCHAR(64) NOT NULL,
   lastName  VARCHAR(64) NOT NULL,
   pin       INTEGER NOT NULL,
   sex       VARCHAR(1),
   height    FLOAT,
   email     VARCHAR(64),
   alive     CHAR(1),
   income    INTEGER,
   dob       SMALLINT,
   CONSTRAINT demo_user_pk PRIMARY KEY (firstName, lastName, pin)
);
```

The last line defines the composite primary key on the `demo_user` table. The primary key columns should be defined as key fields of the mapping between the `User` class and the `demo_user` table. The column SQL types are mapped to Java types as suggested by Oracle. If you are using a different RDBMS product, please refer to the vendor documentation for Java-SQL type mapping details.

Let us look at the primary key class, `UserPK`:

```
public class UserPK {
   // Define all key attribute ouf User class
   String firstName;
   String lastName;
   long pin;
```

```
   // The no-argument constructor, required by dbMapper
   public UserPK() { }
   // Other constructors
   ......
   // Getter/Setter methods for the attributes
   ......
}
```

Note that `UserPK` looks like a stripped version of the `User` class. The only fields that remain are those that compose the primary key. For the user-defined primary key classes, the dbMapper package recommends that the user override the `equals` and `hashCode` methods of the `java.lang.Object` super class.

The `examples/ex03-primary_key_class/data/or_mapping.xml` file defines the mapping between the `User` class and the `demo_user` table:

```
<?xml version="1.0"?>
<!DOCTYPE mappings PUBLIC "DBMapper OR Mapping"
"http://www.onsd.nec.com/software/db_or_mapping.dtd">

<mappings>

<!-- Bind User class to demo_user table. Define primary key class -->
<mapping class = "User"
       table="demo_user"
       pk_class = "UserPK"
      >
      <!-- bind java String firstName/lastName to sql column
           firstName/lastName (indicate as key attribute -->
      <field id="firstName" is_key="true" >
                <basic_type>String</basic_type>
      </field>
      <field id="lastName" is_key="true" >
                <basic_type>String</basic_type>
      </field>
      <!-- bind java long pin to sql column pin:int (indicate as key
      attribute -->
      <field id="pin" is_key="true" >
                <basic_type>long</basic_type>
      </field>

      <!-- define non-key attributes -->
      <field id="sex" is_key="false" >
            <basic_type> char </basic_type>
      </field>
      <field id="height" is_key="false" >
            <basic_type> double </basic_type>
      </field>
      <field id="email" is_key="false" >
            <basic_type> String </basic_type>
      </field>
      <field id="alive" is_key="false" >
            <basic_type> boolean </basic_type>
      </field>
      <field id="income" is_key="false" >
            <basic_type> Integer </basic_type>
      </field>
      <field id="dob" is_key="false" >
            <basic_type> Short </basic_type>
      </field>
</mapping>

</mappings>
```

We are all set. Let us go through the example code to see how `User` objects can be created, deleted, modified, and located in the database and how to make use of the `UserPK` primary key class.

```
public class Test {
   DBInterface mapper; // mapper used for this test

   Test (String mapperFile) {
      // create mapper to be used for this test
      ..............
   }
   public static void main (String[] args) {
      try {
         String mapperFile = args[0];
         Test testSuite = new Test(mapperFile);
         testSuite.test();
      }
      catch (Exception e) {
         Logger.error ("Error details: ", e);
      }
   }

   public void test() throws Exception {
      AttrValMap aValMap = null;
      HashMap    hValMap = null;

      // Delete older entries
      mapper.deleteAll (User.class);

      // Create a User object and persist it to database
      User usr1 = new User ("Charles", "Smith", 13452,
                            'M', 170.34, "csmith@mailcity.com",
                            true, new Integer(55000), new Short((short)15));
      mapper.create (usr1);
      Logger.debug ("create(Object) :: Object " + usr1
                    + " is successfully created.");

      // Now delete the user created above using primay key
      UserPK usr1PK = new UserPK("Charles", "Smith", 13452);
      mapper.deleteByPrimaryKey (usr1PK, User.class);
      Logger.debug ("delete(Object) :: Object " + usr1
                    + " is successfully deleted.");

      // Create usr1 again. Delete all the User objects with last name "Smith"
      // i.e. (deletes usr1 from database)
      mapper.create (usr1);
      Logger.debug ("create(Object) :: Object " + usr1
                   + " is successfully created.");
      aValMap = new AttrValMap();
      aValMap.put ("lastName", "Smith");
      mapper.deleteByAttributes (aValMap, User.class);
      Logger.debug ("deleteByAttributes(aval,cls) :: Object " + usr1
                    + " is successfully deleted.");

      // Create the usr1 again. Update usr1 attributes and update new changes
      // to database
      mapper.create (usr1);
      Logger.debug ("create(Object) :: Object " + usr1
                   + " is successfully created.");
      usr1.setDob (new Short((short) 18));
      usr1.setEmail (null);
      usr1.setIncome (null);
      mapper.update(usr1);
      Logger.debug ("update(Object) :: Object " + usr1
                   + " is successfully updated.");

      // Update selected attributes (using AttrValMap)
      // of usr1 and update these new changes to database
      aValMap = new AttrValMap();
      aValMap.put ("dob", new Short((short) 12));
      aValMap.put ("income", new Integer(67759));
      aValMap.put ("email", null);
      mapper.update(usr1, aValMap, true);
      Logger.debug ("update(Object,aval,bool) :: Object " + usr1
                   + " is successfully updated.");
```

```
// Update selected attributes (using HashMap) of usr1 and update
// these new changes to database
hValMap = new HashMap(1);
hValMap.put ("income", new Integer(23234));
hValMap.put ("dob", new Short((short)30));
hValMap.put ("email", "smith@hotmail.com");
mapper.update(usr1, hValMap, true);
Logger.debug ("update(Object,hval,bool) :: Object " + usr1
           + " is successfully updated.");

// Create few more users
User usr2 = new User ("Goldy", "Smith", 786, 'F', -1,
                      null, false, new Integer(1000000),
                      new Short((short)24));
mapper.create (usr2);
User usr3 = new User ("Kate", "Winslet", 1234, 'F', 168.23,
                     "katie@hollywood.com", true, new Integer(999699),
                     new Short((short)12));
mapper.create (usr3);
User usr4 = new User ("Princess", "Diana", 666, 'F', 178,
                      "diana@celebs.com", false, new Integer(666978),
                      new Short((short)6));
mapper.create (usr4);

// Locate an previously created user in database using primary key
UserPK usr2PK = new UserPK ("Goldy", "Smith", 786);
User usr2dup = (User) mapper.findByPrimaryKey (
                           usr2PK, User.class); // Should return usr2
Logger.debug ("findByPrimaryKey(pk,class) :: Object " + usr2dup
             + " is successfully found.");

// Locate an previously created user in database using key attribues
aValMap = new AttrValMap();
aValMap.put ("firstName", "Princess");
aValMap.put ("lastName", "Diana");
aValMap.put ("pin", 666);
Collection c1 = mapper.findByAttributes (aValMap, User.class);
// c1 should contain only usr4
Logger.debug ("findByAttributes(aval,class) :: Collection "
           + c1.toString() + " is successfully found.");

// Find all the alive females from database
aValMap = new AttrValMap();
aValMap.put ("sex", 'F');
aValMap.put ("alive", true);
Collection c2 = mapper.findPrimaryKeysByAttributes (aValMap,
                                                    User.class);
Logger.debug ("findByAttributes(aval,class) :: Collection "
             + c2.toString() + " is successfully found.");
// Collection c2 should only contain primary object pointing to usr3.
// Use the primary key to delete this entry from database
Object akey = c2.iterator().next();
mapper.deleteByPrimaryKey (akey, User.class);
Logger.debug ("deleteByPrimaryKey(key,class) :: Object " + akey
             + " is successfully deleted.");

// Find User objects based on a custom SQL query
Collection c3 = mapper.findByQuery (
              "SELECT * from demo_user where lastName='Smith'",
              User.class);
Logger.debug ("findByQuery(query,class) :: Collection " + c3.toString()
            + " is successfully found.");

// Find all the User objects stored in demo_user table
Collection c4 = mapper.findAll (User.class);
Logger.debug ("findAll(class) :: Collection " + c4.toString()
             + " is successfully found.");

// Find a primary key object based on non-key attribute sex.
// (find all men from database)
```

```
      aValMap = new AttrValMap();
      aValMap.put ("sex", 'M');
      Collection c5 = mapper.findPrimaryKeysByAttributes (
                                                  aValMap, User.class);
      Logger.debug ("findPrimaryKeysByAttributes(aval,class) :: Collection "
                   + c5.toString() + " is successfully found.");

      // Find primary key objects based on a custom SQL query
      Collection c6 = mapper.findPrimaryKeysByQuery (
          "SELECT pin, lastName, firstName FROM demo_user WHERE lastName"
          + " like 'Sm%' ORDER by firstName, lastName, pin",
          User.class);
      Logger.debug ("findPrimaryKeysByQuery(query,class) :: Collection "
                  + c6.toString() + " is successfully found.");

      // Find primary key of all the User objects stored in
      // demo_user table
      Collection c7 = mapper.findAllPrimaryKeys (User.class);
      Logger.debug ("findAllPrimaryKeys(class) :: Collection "
                  + c7.toString() + " is successfully found.");

      // Add usr3 ("Kate Winslet") back.
      mapper.create (usr3);
      Collection c8 = mapper.findAll (User.class);
      Logger.debug ("findAll(class) :: Collection " + c8.toString()
                   + " is successfully found.");
   }
}
```

4.4.5 Example4 – Transaction

This example demonstrates management of transaction boundaries across a set of `DBInterface` method invocations. We are going to use the `Primitive` class and the mapping defined in the "Example2 – Basic Type" section.

The first part of the following code snippet creates a new transaction for the current thread (by invoking the `beginTransaction()` method), performs some successful database operations, and finally terminates the transaction by committing all the database changes that were made (using the `commitTransaction()` method).

The second part of the example code creates a new transaction for the current thread, then performs some valid database operations followed by an error-prone operation (i.e. a database constraint violation). As a result of the bad operation, an exception is thrown by the dbMapper package and the code rolls back all of the database changes that had been made within the transaction.

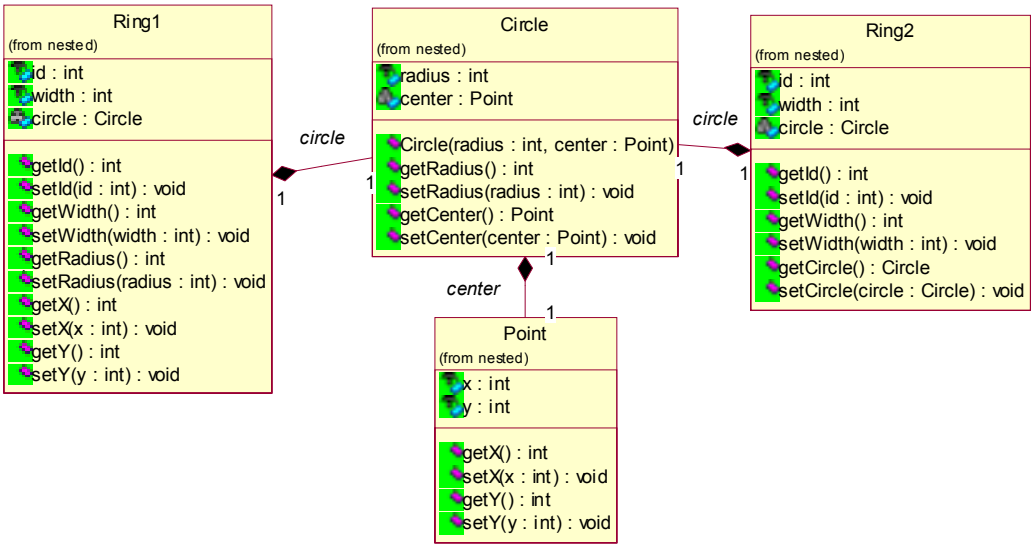
```
   void test() throws Exception {
      AttrValMap aValMap = null;
      HashMap    hValMap = null;

      Logger.debug ("cleanup");
      // Delete older entries
      mapper.deleteAll (Primitive.class);
      display ();

      // Create few Primitive objects and persist them to database
      Primitive o1 = new Primitive(1,null);
      Primitive o2 = new Primitive(2, "Hey");
      Primitive o3 = new Primitive(3, "Welcome !");
      mapper.create (o1);
      mapper.create (o2);
      display ();

      // Demonstrate successful transaction commit
      // Create a new transaction for current thread
      mapper.beginTransaction();
      Logger.debug ("demonstrating transaction commit.");
      try {
```

```
      // Perfrom some valid database operations
      mapper.create (o3);
      mapper.delete (o2);
      o3.setY ("New value");
      mapper.update (o3);
      // Commit all database changes
      mapper.commitTransaction();
   }
   catch (Exception ex1) {
      // Should not happen, just in case (dump the error msg and
      // terminate the process)
      Logger.debug ("failure: Unexpected exception : " + ex1);
      System.exit(1);
   }
   Logger.debug ("sucess. the values after modification.");
   display ();

   // Demonstrating unsuccessful transaction (rollback case)
   // Create a new transaction for current thread
   mapper.beginTransaction();
   Logger.debug ("demonstrating transaction rollback.");
   try {
      // Perfrom some valid database operations
      mapper.create (o2);
      mapper.delete (o3);
      // Bad operation. Trying to re-insert o1 which will fail as
      // it is a duplicate record (primary key violation)
      mapper.create (o1);
      // Code should never reach here, if it does, dump the error
      // msg and terminate the process
      Logger.debug ("failure: Unexpected error.");
      System.exit(1);
   }
   catch (Exception ex1) {
      // Rollback the database changes made in this transaction
      mapper.rollbackTransaction();
   }
   // Dump all the records and verify changes made in above transaction
   // are rolled back (e.g. o3 is not deleted)
   Logger.debug ("sucess. the values after rollback.");
   display ();
}

private void display () throws Exception {
   Collection c = mapper.findAll (Primitive.class);
   Logger.debug ("Entries in database: " + c.toString());
}
```

4.4.6 Example5 – Nested Field

Now that you have seen how the dbMapper package can persist user objects that contain only basic fields to a relation database, it is time to explore some of the more advanced OR mapping concepts defined by the dbMapper package, such as nested fields. In this example, we will define two analogous classes, namely `Ring1` and `Ring2`. Each of, these classes contain three nested Java attributes, namely `circle.radius`, `circle.center.x` and `circle.center.y` (please see the containment relationship described in the class diagram below). These two classes are mapped to the `demo_ring` table. We will present two alternatives to map these nested Java fields to the `demo_ring` table columns:

- **`Ring1` class mapping**: Map the nested attributes as basic fields by providing custom get and set methods for the nested attributes.
- **`Ring2` class mapping**: Map the nested attributes directly as nested_fields (as described in the "nested_type Element" section).



The SQL schema for `demo_ring` table:

```
create table demo_ring (
   id INTEGER PRIMARY KEY,
   radius INTEGER,
   width INTEGER,
   x INTEGER,
   y INTEGER
)
```

The `circle.radius`, `circle.center.x` and `circle.center.y` nested fields are mapped to the `radius`, `x` and `y` columns, respectively.

To demonstrate the first mapping alternative, let us look at the `Ring1` class definition and it's mapping to the `demo_ring` table:

```
public class Ring1 {
   // Attribute declarations
   ....
   // The no-argument constructor, reqd by dbMapper
   public Ring1() {
      // Created and initialize all nested fields
      circle = new Circle(0, new Point(0,0));
   }
   // Getter and setter methods
   .....
   public int getX () {
      return circle.getCenter().getX();
   }
   public void setX (int x) {
      circle.getCenter().setX (x);
   }
   ....
}

<mapping class = "Ring1" table="demo_ring" >
       .......
      <field id="x">
                 <basic_type>int</basic_type>
      </field>
       .......
</mapping>
```

This class definition and class mapping demonstrate how a nested attribute, in this case the `circle.center.x` attribute, may be mapped as a basic field. This is accomplished by supplying the custom set and get methods, `getX()` and `setX()`, which access the nested attribute directly (i.e. without referencing any intermediate object).

The second alternative provides a cleaner method for mapping nested attributes to database columns. This method does not require the mapped class to define the custom set and get methods, e.g. `getX()`and `setX()`. Before going through this example, please be sure to read the example covered in the "nested type Element" section.

The following code snippet shows how the `circle.center.x` nested Java field of the `Ring2` class is mapped as a nested field to the column `x`:

```
<mapping class = "Ring2" table="demo_ring" >
       .......
      <field id="x">
                 <nested_type column="x">
               <intermediate_node node_id="circle" class="Circle" />
               <intermediate_node node_id="point"  class="Point">
                  <get_method> getCenter </get_method>
               </intermediate_node>
               <leaf_node node_id="x" class="int" />
            </nested_type>
      </field>
       .......
</mapping>
```

The nested `x` field of a `Ring2` object, say `ring2`, is accessed as follows:

```
ring2.getCircle().getCenter().getX()
```

To modify this nested attribute, dbMapper will use:

```
ring2.getCircle().getCenter().setX(newIntegerValue)
```

If any of the intermediate get methods return a `null` object, the dbMapper acts as if the leaf field, `x`, was `null`.

Notice that the default `Ring2` constructor (i.e. the constructor that takes no arguements) creates all of the intermediate objects in the `Ring2` object containement tree:

```
public class Ring2 {
   ....
   // The default constructor, reqd by dbMapper
   public Ring2() {
      // Created and initialize all nested fields
      circle = new Circle(0, new Point(0,0));
   }
   ...
}
```

Now it is time to create and persist some `Ring1` and `Ring2` objects to the database. The following code is taken from the `Test.java` file:

```
   // Test Ring1 (nested attributes mapped to basic types)
   void testRing1 () throws Exception {
      // Delete older entries
      mapper.deleteAll (Ring1.class);

      // Create new entries
      Ring1 r1 = new Ring1 (1, 3, new Circle(9, new Point(1,2)));
      Ring1 r2 = new Ring1 (2, 4, new Circle(16, new Point(2,3)));
      mapper.create (r1);
```

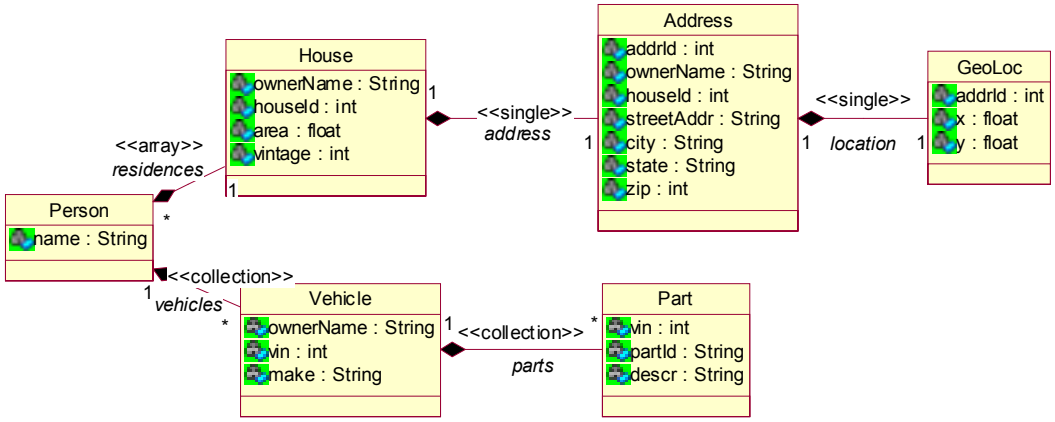
```
   mapper.create (r2);
   Logger.debug ("create(Object) :: Object " + r1 + " and " + r2
               + " is successfully created.");

   // Update some nested attributes
   AttrValMap aMap1 = new AttrValMap();
   aMap1.put ("x", 5);
   aMap1.put ("radius", 11);
   mapper.update (r1, aMap1, true);
   Logger.debug ("update(object,aMap,bollean) on object " + r1
               + " is successfully excuted.");

   // Find objects by giving nested attributes
   AttrValMap aMap2 = new AttrValMap();
   aMap2.put ("y", 3);
   Collection c1 = mapper.findByAttributes (aMap2, Ring1.class);
   Logger.debug ("findByAttributes(aval,class) :: Collection "
                 + c1.toString() + " is successfully found.");
}


// Test Ring2 (nested attributes mapped to nested types)
void testRing2 () throws Exception {
   // Delete older entries
   mapper.deleteAll (Ring2.class);

   // Create new entries
   Ring2 r1 = new Ring2 (1, 2, new Circle(32, new Point(0,0)));
   Ring2 r2 = new Ring2 (2, 2, new Circle(25, new Point(1,1)));
   mapper.create (r1);
   mapper.create (r2);
   Logger.debug ("create(Object) :: Object " + r1 + " and " + r2
               + " is successfully created.");

   // Create a ring which has null circle (intermediate nested attributes
   // are null)
   Ring2 r3 = new Ring2 (3, 3, null);
   mapper.create(r3);
   Logger.debug ("create(Object) :: Object " + r3
               + " is successfully created.");

   // Update some nested attributes
   AttrValMap aMap1 = new AttrValMap();
   aMap1.put ("x", 3);
   aMap1.put ("radius", 21);
   mapper.update (r1, aMap1, true);
   Logger.debug ("update(object,aMap,bollean) on object " + r1
               + " is successfully excuted.");

   // Find objects by giving nested attributes
   AttrValMap aMap2 = new AttrValMap();
   aMap2.put ("y", 1);
   Collection c1 = mapper.findByAttributes (aMap2, Ring2.class);
   Logger.debug ("findByAttributes(aval,class) :: Collection "
                 + c1.toString() + " is successfully found.");

   // Load ring with no circle.
   Ring2 r3dup = (Ring2) mapper.findByPrimaryKey (new Integer(3),
                                                  Ring2.class);
   Logger.debug ("findByPrimaryKey(key,class) :: found object " + r3dup);
}
```

4.4.7 Example6 – Person Class (Complex and Complex Collection Fields)

All of the examples discussed so far use classes that are simple in the sense that each of their fields can be mapped to a single database column. This example demonstrates the use of more complicated classes, such as those that contain instances of other user-defined classes, or collections of objects.

Note that the dbMapper supports both one-to-one and one-to-many relationships between a user object and its fields. Please refer to the "Field Types" section for a more detailed discussion of the complex and complex collection fields.

The containment relationship used in this example is as follows:



A `person` object owns several `houses` (many-to-one array relation) and `vehicles` (many-to-one collection relation). Each `house` has one `address` (one-to-one relation) whose geographic location is specified by a `GeoLoc` object (one-to-one relation). A `vehicle` object may contain several `parts` (many-to-one collection relation).

Let us look at the class definitions:

```
public class Person {
   // Primary key attribute
   String  name;
   // many-to-one array relationship with residences attribute
   House[] residences;
   // many-to-one collection relationship with vehicles attribute
   Vector  vehicles; // Vector of Vehicle

   // The no-argument constructor, reqd by dbMapper
   public Person() { }
   // Other constructors
   ......

   // Getter/Setter methods for the fields
   public String getName () { return name; }
   public House[] getHouses () { return residences; }
   public Vector getVehicles () { return vehicles; }

   public void setName (String name) { this.name = name; }
   public void setHouses (House[] residences) { this.residences = residences; }
   public void setVehicles (Vector vehicles) { this.vehicles = vehicles; }
   // Other methods
   ......
}

public class House {
   // Primary key attributes
   String  ownerName;
   int     houseId;
   // one-to-one relationship with address attribute
   Address address;
   // other (basic) attributes
   float   area;
   int     vintage;
```

```
   .........
}

public class Address {
   // Primary key attribute
   int     addrId;
   // Parent class relation attributes
   String  ownerName;
   int     houseId;
   // one-to-one relationship with location attribute
   GeoLoc  location;
   // other (basic) attributes
   String  streetAddr;
   String  city;
   String  state;
   int     zip;
   .........
}

public class GeoLoc {
   // Primary key attribute
   int     addrId;
   // other (basic) attributes
   float   x;
   float   y;
   .........
}

public class Vehicle {
   // Primary key attribute
   int     vin;
   // many-to-one collection relationship with parts attribute
   ArrayList parts = new ArrayList(1);
   // Parent class relation attributes
   String  ownerName;
   // other (basic) attributes
   String  make;
   .........
}

public class Part {
   // Primary key attribute
   String  partId;
   // Parent class relation attributes
   int     vin;
   // other (basic) attributes
   String  descr;
   .........
}
```

The basic fields of the `Person` class are mapped to the `demo_person` table columns as follows:

```
create table demo_person (
   name VARCHAR(64) PRIMARY KEY
);
```

The `demo_person_house` table is used to store the `residences` attribute of a `Person` object. Note the foreign key relationship between the `demo_person.name` and `demo_person_house.ownerName` columns. Multiple `demo_person_house` records may be linked to a single `demo_person` record using this foreign key relationship; and upon deletion of the `demo_person` record, all these `demo_person_house` records are automatically deleted. The `ownerName` column is also part of the composite primary key defined on the `demo_person_house` table.

```
create table demo_person_house (
   ownerName VARCHAR(64) NOT NULL,
   houseId INTEGER NOT NULL,
```

```
   area FLOAT,
   vintage INTEGER,
   CONSTRAINT constr_demo_ph_pk PRIMARY KEY(ownerName, houseId),
   CONSTRAINT constr_demo_ph_fr_name FOREIGN KEY(ownerName)
                           REFERENCES demo_person(name) ON DELETE CASCADE
)
```

The `house address` is saved in the `demo_person_address` table. This table has a single primary key column named `addrId`. The `houseId` and `ownerName` columns capture the one-to-one relationship between a `demo_person_house` and `demo_person_address` database record.

```
create table demo_person_address (
   addrId    INTEGER PRIMARY KEY,
   ownerName VARCHAR(64),
   houseId INTEGER,
   street VARCHAR(128),
   city VARCHAR(64),
   state VARCHAR(32),
   zip INTEGER,
   CONSTRAINT constr_demo_pa_fr_house FOREIGN KEY (ownerName, houseId)
              REFERENCES demo_person_house(ownerName, houseId) ON DELETE CASCADE
)
```

The `demo_person_address_location`, `demo_person_vehicle` and `demo_person_vehicle_part tables` are mapped to the `GeoLoc`, `Vehicle`, and `Part` classes respectively. The table schema (constraints, key relations, primary key etc.) follows the same conventions as described above:

```
create table demo_person_address_location (
   addrId  INTEGER NOT NULL,
   x   FLOAT,
   y   FLOAT,
   CONSTRAINT constr_demo_pal_fr_id FOREIGN KEY (addrId)
              REFERENCES demo_person_address(addrId) ON DELETE CASCADE
)

create table demo_person_vehicle (
   vin     INTEGER PRIMARY KEY,
   ownerName VARCHAR(64) NOT NULL,
   make VARCHAR(128),
   CONSTRAINT constr_demo_pv_fr_name FOREIGN KEY(ownerName)
                           REFERENCES demo_person(name) ON DELETE CASCADE
)

create table demo_person_vehicle_part (
   partId  VARCHAR(64) PRIMARY KEY,
   vin     INTEGER NOT NULL,
   descr VARCHAR(128),
   CONSTRAINT constr_demo_pvp_fr_vin FOREIGN KEY(vin)
                  REFERENCES demo_person_vehicle(vin) ON DELETE CASCADE
)
```

The following is the mapping file used to associate the `Person`, `House`, `Address`, `GeoLoc`, `Vehicle` and `Part` classes to the corresponding database tables, which were described above:

```
<?xml version="1.0"?>
<!DOCTYPE mappings PUBLIC "DBMapper OR Mapping"
"http://www.onsd.nec.com/software/db_or_mapping.dtd">

<mappings>

<mapping class = "Person"  table="demo_person" >
      <!-- implicit primary key class "String" -->
      <field id="name" is_key="true" >
                                  <basic_type>String</basic_type>
      </field>
```

```
      <field id="residences" is_key="false" >
                <get_method> getHouses </get_method>
                <set_method> setHouses </set_method>
                   <complex_collection_type>
                      <element_mapref class="House"/>
                      <!-- more than one house stored in an array. -->
                      <container_class> House[] </container_class>
                      <key_bindings>
                         <key_binding parent_field="name" child_field="ownerName"/>
                      </key_bindings>
                   </complex_collection_type>
      </field>
      <field id="vehicles" is_key="false" >
                   <complex_collection_type>
                      <element_mapref class="Vehicle"/>
                      <!-- more than one vehicle stored in a vector(collection). -->
                      <container_class>java.util.Vector</container_class>
                      <key_bindings>
                         <key_binding parent_field="name" child_field="ownerName"/>
                      </key_bindings>
                   </complex_collection_type>
      </field>
</mapping>

<mapping class = "House" table="demo_person_house" >
      <!-- pk_class name="String,int" -->
      <field id="ownerName" is_key="true" >
                                 <basic_type>String</basic_type>
      </field>
      <field id="houseId" is_key="true" >
                                  <basic_type>int</basic_type>
      </field>
      <field id="area" is_key="false" >
                                  <basic_type>float</basic_type>
      </field>
      <field id="vintage" is_key="false" >
                                  <basic_type>int</basic_type>
      </field>
      <field id="address" is_key="false" >
                   <complex_type>
                      <element_mapref class="Address"/>
                      <key_bindings>
                         <key_binding parent_field="ownerName" child_field="ownerName"/>
                         <key_binding parent_field="houseId" child_field="houseId"/>
                      </key_bindings>
                   </complex_type>
      </field>
</mapping>

<mapping class = "Address"
       table="demo_person_address" >
      <!-- pk_class name="int" -->
      <field id="addrId" is_key="true" >
                                  <basic_type>int</basic_type>
      </field>
      <field id="ownerName" >
                                 <basic_type>String</basic_type>
      </field>
      <field id="houseId" >
                                  <basic_type>int</basic_type>
      </field>
      <field id="streetAddr" is_key="false" >
                         <basic_type column="street">String</basic_type>
      </field>
      <field id="city" is_key="false" >
                                 <basic_type>String</basic_type>
      </field>
      <field id="state" is_key="false" >
                                 <basic_type>String</basic_type>
      </field>
      <field id="zip" is_key="false" >
```

```
                                    <basic_type>int</basic_type>
      </field>
      <field id="location" is_key="false" >
                  <complex_type>
                     <element_mapref class="GeoLoc"/>
                     <key_bindings>
                        <key_binding parent_field="addrId" child_field="addrId"/>
                     </key_bindings>
                  </complex_type>
      </field>
</mapping>

<mapping class = "GeoLoc"
       table="demo_person_address_location" >
      <!-- no pk_class -->
      <field id="addrId" is_key="false" >
                                     <basic_type>int</basic_type>
      </field>
      <field id="x" is_key="false" >
                                    <basic_type>float</basic_type>
      </field>
      <field id="y" is_key="false" >
                                    <basic_type>float</basic_type>
      </field>
</mapping>

<mapping class = "Vehicle"
       table="demo_person_vehicle" >
      <!-- pk_class name="int" -->
      <field id="vin" is_key="true" >
                                     <basic_type>int</basic_type>
      </field>
      <field id="ownerName" >
                                   <basic_type>String</basic_type>
      </field>
      <field id="make" is_key="false" >
                                   <basic_type>String</basic_type>
      </field>
      <field id="parts" is_key="false" >
                  <complex_collection_type>
                     <element_mapref class="Part"/>
                     <!-- more than one part stored in a list(collection). -->
                     <container_class>java.util.ArrayList</container_class>
                     <key_bindings>
                        <key_binding parent_field="vin" child_field="vin"/>
                     </key_bindings>
                  </complex_collection_type>
      </field>
</mapping>

<mapping class = "Part"
       table="demo_person_vehicle_part" >
      <!-- int pk_class -->
      <field id="partId" is_key="true" >
                                   <basic_type>String</basic_type>
      </field>
      <field id="vin" is_key="false" >
                                     <basic_type>int</basic_type>
      </field>
      <field id="descr" is_key="false" >
                                   <basic_type>String</basic_type>
      </field>
</mapping>

</mappings>
```

Note that the `Person` OR mapping overrides the default get/set method names for the `residences` field. Please refer to the "field Element" section for details. Now we are ready to use the above mapping to create a `Person` object (with complex and complex collection fields) in the database.

```
class TestPerson {
   // create mapper object
   ................

   public void test() throws Exception {
      // Delete older entries
      mapper.deleteAll (Person.class);

      // Instantiate all House objects owned by "Charles Smith"
      Address r1 = new Address (1, "Charles Smith", 101, "12056 Greywing Sq",
                                "Reston", "VA", 20191,
                                new GeoLoc(1, (float)34.5, (float)-23.6));
      House   h1 = new House ("Charles Smith", 101, 1024, 5, r1);
      Address r2 = new Address (2, "Charles Smith", 102, "13452 Farmcrest Ct",
                                "Herndon", "VA", 20171,
                                new GeoLoc(1, (float)34.45, (float)-23.61));
      House   h2 = new House ("Charles Smith", 102, 1025, 1000000, r2);

      // Instantiate all the vehicles owned by "Charles Smith"
      Vehicle v1 = new Vehicle ("Charles Smith", 234567, "Toyota Camary", null);
      ArrayList parts1 = new ArrayList();
      parts1.add (new Part(874687, "steering",
                           "steer the vehicle in desired direction."));
      parts1.add (new Part(874687, "tyres", null));
      Vehicle v2 = new Vehicle ("Charles Smith", 874687, "Ford", parts1);

      // Instantiate the "Charles Smith" Person object with above
      // houses and vehicles
      House[] houses1 = new House[] {h1,h2};
      Vector  vehicles1 = new Vector(2);
      vehicles1.add (v1);
      vehicles1.add (v2);
      Person p1 = new Person("Charles Smith", houses1, vehicles1);

      // Persist the entire Person object containment tree (including
      // vehicles, houses, address, locations, parts) in database
      mapper.createTree (p1);
      Logger.debug ("create(Object) :: Object " + p1
                  + " is successfully created.");
      Logger.debug ("-------------------------------------------------------");

      // Find the persistent "Charles Smith" Person object in database
      // Load the entire Person object containment tree from database
      // in another Person object
      Person p1Dup = (Person) mapper.findByPrimaryKey ("Charles Smith",
                                                       Person.class, 9999);
      // Compare p1 and p1Dup object containment tree in the log file,
      // field by field.  The fields should have same value.
      Logger.debug ("findByPrimaryKey() :: Object " + p1Dup
                  + " is successfully executed.");
   }
}
```

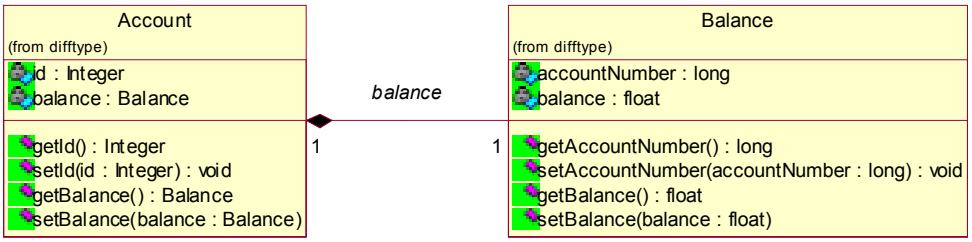
Refer to the "Using a DBInterface" section for more examples of database operations on an object with complex and complex collection fields.

4.4.8 Example7 - Key Binding Field Types

As part of demonstrating how the dbMapper handles complex and complex collection fields, the example in the previous section demonstrated the use of key bindings. Recall that the key bindings are used by the dbMapper to correlate complex and complex collection fields to their containing objects. In the example in the previous section, the Java types and SQL types of the parent and child fields of the key bindings were the same. Although such consistency between the parent and child fileds of the key binding is recommended, it is not necessary. This section provides an example that shows that the parent and child fields of the key binding need not necessarily be of the same Java and SQL types.

Before presenting the example, we first state the only restriction that applies to the types of the child and parent fields of a key binding: either the Java type of the child field can be promoted by the Java compiler to the type of the parent field, or vice versa. Note that there are no restrictions or dependencies between the SQL types of the child and parent fields of the key binding. (Of course, the SQL type used to store any field must be consistent with the Java type of the field.)

In this example, an `Account` class contains a complex field named `balance`, of type `Balance`. The key binding used by this example associates the `accountNumber` field of the `Balance` class with the `id` field of the `Account` class. Note that the Java type of the `accountNumber` field is `long` and its SQL type is `INTEGER`, while the Java type of the `id` field of the `Account` class is `Integer` and its SQL type is `SMALLINT`.



The `Account` class uses the `demo_account` table (for storage of the basic fields), and the `Balance` class uses the demo_balance `table`.

```
create table demo_account (
   id SMALLINT PRIMARY KEY
)
create table demo_balance (
   accountNumber INTEGER NOT NULL,
   balance FLOAT,
   CONSTRAINT constr_demo_ch_fr_accountnum FOREIGN KEY(accountNumber)
            REFERENCES demo_account(id) ON DELETE CASCADE
)
```

The class mappings for the `Account` and `Balance` class follow:

```
<mapping class = "Account" table="demo_account" >
      <!-- implicit primary key class "Integer" -->
      <field id="id" is_key="true" >
                <basic_type column="id">Integer</basic_type>
      </field>
      <field id="balance" is_key="false" >
                   <complex_type>
                      <element_mapref class="Balance"/>
                      <key_bindings>
                         <key_binding parent_field="id" child_field="accountNumber"/>
                      </key_bindings>
                   </complex_type>
      </field>
</mapping>

<mapping class = "Balance"   table="demo_balance" >
      <field id="accountNumber" is_key="true">
                <basic_type>long</basic_type>
      </field>
      <field id="balance" >
                <basic_type>float</basic_type>
      </field>
</mapping>
```

The following code fragment simply creates and persists a new `Account` object containment tree to the database. Later it reads the entire `Account` object containment tree from the database to memory.

```
   void test() throws Exception {
      // Delete older entries
      mapper.deleteAll (Account.class);

      Account p1 = new Account (new Integer(1), new Balance(1, (float) 430.35));
      mapper.createTree (p1);

      Account p1Dup = (Account) mapper.findByPrimaryKey (new Integer(1),
                                                         Account.class, 3);
   }
```

4.4.9 EJB Example

The examples discussed in this section introduce another powerful feature of the dbMapper: how to write bean managed persistent (BMP) for an entity bean using dbMapper.

4.4.9.1 Compiling and Running This Example

Since an EJB example, which is located in the "ex08-ejb" directory, is substantially more involved than the other examples, the procedures to compile and run it are somewhat different from the other examples. You will need to have "Ant" installed on your system to compile this example. Also, note that this example was written for the Orion application server. To compile the example with other application servers, you may need to modify the "env.bat" and "build.xml" files accordingly. (It is a good idea to backup the original files before modifying them with your changes.)

Before compiling and running the example, you will first need to modify the "env.bat" file to match your setup, and then execute it toget the desired environment.

The "build.xml" file contains the instructions to compile the example. First edit that file and modify any settings as needed to match your setup. Once the file is modified, run "ant" to compile the example.

Before running the example, make sure that your database and application server are up and running. Then execute the "run.bat" script to run the example.

4.4.9.2 Counter Entity Bean

Our EJB example will be a simple counter bean. The counter bean represents a dynamic counter value. Through persistence, the counter value is stored in an underlying relational database.

Let's look at the counter bean remote interface, which exposes methods for incrementing and decrementing the counter value:

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

public interface Counter extends EJBObject {
   // Increment the counter by 1
   public int increment() throws RemoteException;
   // Decrement the counter by 1
   public int decrement() throws RemoteException;
   // Get the current counter value
   public int value() throws RemoteException;
}
```

The home interface for `Counter` is specified by `CounterHome.java`, shown below. The `CounterHome` class defines a single factory method, `create()`, to create `Counter` EJB objects:

```
// import statements

public interface CounterHome extends EJBHome {
   // creates a Counter EJB object with given (unique) counter identifier.
```

```
   public Counter create(String counterId) throws CreateException, RemoteException;
   // Find a counter by its primary key (counter id).
   public Counter findByPrimaryKey(String primaryKey)
                                 throws FinderException, RemoteException;
   // Returns all the Counter entity beans stored in database
   public Enumeration findAll() throws FinderException, RemoteException;
   // Returns all the Counter beans that have a non-zero value
   public Enumeration findNonZeroCounters() throws FinderException, RemoteException;
}
```

The `create()` method creates a new database record representing a counter. The `CounterHome` interface defines three finder methods. The `findByPrimaryKey` method searches the database for a counter that already exists. The `findAll` method returns all counters stored in the database. The `findNonZeroCounters` method searches the database for counters that have a non-zero counter value.

The Counter entity bean's primary key (counter `id`) is a simple `String` object. The client code that constructs the counter `id` should make sure that it's unique.

Before going through the entity bean implementation class, `CounterEJB`, let's look at the helper class, `CounterBean`, and the `demo_counter` database table. The `demo_counter` database table consists of two columns: `id` and `value.` The `id` column is the primary key for this table..

```
create table demo_counter (
   id VARCHAR(64) PRIMARY KEY,
   value INTEGER
)
```

`CounterBean` is a simple Java class that encapsulates all the necessary information for a counter. The `CounterBean` class is composed of two fields, `counterId` and `value`:

```
public class CounterBean {
   private String counterId; // Holds counter bean's primary key (counter id)
   private int value = 0;    // Holds current counter value (initialized with 0)

   public CounterBean () {
   }
   // Getter & setter methods for counterId & value fields
   ...
   public int increment() {
      return ++value;
   }
   public int decrement() {
      return --value;
   }
}
```

The class mapping (between the `CounterBean` class and `demo_counter` table) is specified by the following OR mapping file (with path `data/or_mapping.xml`):

```
<?xml version="1.0"?>
<!DOCTYPE mappings PUBLIC "DBMapper OR Mapping"
"http://www.onsd.nec.com/software/db_or_mapping.dtd">

<mappings>
<mapping class = "CounterBean"
       table="demo_counter" >
      <!-- implicit primary key class "String" -->
      <field id="counterId" is_key="true" >
                <basic_type column="id">String</basic_type>
      </field>
      <field id="value" is_key="false" >
                <basic_type>int</basic_type>
      </field>
</mapping>
```

```
</mappings>
```

The above mapping is very similar to the other mappings discussed in earlier examples. The primary key field, `counterId`, is mapped to the `id` column of the `demo_counter` table. The counter `value` field is mapped to the `value` column. Since the above mapping contains only one key field, `counterId`, the `String` class will be used as the primary key class for the mapping.

This example assumes that the application server where the counter entity bean is deployed is configured with a JNDI data source that can be accessed within EJB implementation code through the JNDI context. The JNDI data source is used by the dbMapper to save and load counter beans to and from a database. The `dbmapper.xml` file located in the `data` directory is used to initialize the `DBMapper` object. Note, absolute paths are used for both the `dbmapper.xml` and `or_mapping.xml` files, as they are loaded from the ejb jar file (see section "DBModule Class"). The content of the `dbmapper.xml` file follows:

```
<?xml version="1.0"?>
<!DOCTYPE root PUBLIC "DBMapper Config" "http://www.onsd.nec.com/software/dbmapper.dtd">

<root>

<data_sources>
   <data_source id="default_ds">
      <jndi_data_source jndi_location="jdbc/OracleCoreDS" />
   </data_source>
</data_sources>

<mapping_contexts>
   <mapping_context id="default" data_source_id="default_ds">
      <or_mapping_files>
         <or_mapping_file path="/or_mapping.xml"/>
      </or_mapping_files>
   </mapping_context>
</mapping_contexts>

</root>
```

The above configuration defines a JNDI data source named `default_ds` that makes use of the javax.sql.DataSource that is bound to the `"jdbc/OracleCoreDS"` JNDI path at the application server. The "default" mapping context is created by binding the `default_ds` data source and the mappings (already discussed) defined in `or_mapping.xml` file.

Our entity bean implementation is specified by the CounterEJB.java class, shown below:

```
import java.io.Serializable;
import com.nec.tdd.tools.dbMapper.*;
import java.util.*;
import javax.ejb.*;

public class CounterEJB implements EntityBean {
```

The above snippet declares the `CounterEJB` class that represents a counter bean. Notice that the `CounterEJB` class extends the `EntityBean` interface, which all entity bean implementations must do. The following code snippet declares two variables, namely `counter` and `mapper`. The `counter` field is the only persistent field of our entity bean class. The `CounterEJB` class will load and store the database data in the `counter` field using the mapper named `mapper`.

```
   // Bean-managed state field
   private CounterBean counter = new CounterBean();
   // The database interface used to load/store counter from/to database.
   private DBInterface mapper = null;
```

The following code declares the `ctx` attribute and related methods (required by the EJB specification). The `ctx` attribute stores the entity bean context and can later be used to acquire the environment information.

```
private EntityContext ctx;
public void setEntityContext(EntityContext ctx) {
   this.ctx = ctx;
}

public void unsetEntityContext() {
   this.ctx = null;
}
```

The EJB container invokes the `ejbLoad` method to load database data into the bean instance. The `ejbLoad` method acquires the primary key via the `getPrimaryKey()` call to the entity bean context. This is done to determine what data should be loaded into the `counter` field. Next, the `initDBIf()` method (explained later in this section) is invoked to initialize the mapper, `mapper`. Subsequently, the `findByPrimaryKey()` method call on `mapper` updates the in-memory entity bean object to reflect the current counter value stored in the database.

```
/**
 * Loads the EJB from the persistent storage.
 */
public void ejbLoad() {
   String id = (String) ctx.getPrimaryKey();
   System.out.println ("ejbLoad (" + id + ")");
   try {
      initDBIf();
      CounterBean newCounter =
               (CounterBean) mapper.findByPrimaryKey (id, CounterBean.class);
      if (null == counter) {
         throw new NoSuchEntityException ("ejbLoad: No counter with id="
                                          + id);
      }
      counter = newCounter;
   } catch (Exception ex) {
      throw new EJBException(ex);
   }
}
```

The EJB container calls the `ejbStore` method to update the database to the current values of this entity bean instance.

```
/**
 * Stores the EJB in the persistent storage.
 */
public void ejbStore() {
   String id = (String) ctx.getPrimaryKey();
   System.out.println ("ejbStore (" + id + ")");
   try {
      initDBIf();
      mapper.update(counter);
   } catch (Exception ex) {
      throw new EJBException(ex);
   }
}
```

The EJB container invokes the following EJB-create method when a client calls the `create(String counterId)` method on a `CounterHome` object. The `ejbCreate` method attempts to add a new counter into the database with the given counter identifier.

```
public String ejbCreate (String counterId) throws CreateException {
   System.out.println ("ejbCreate (" + counterId + ")");
   counter.setCounterId(counterId);
   counter.setValue(0);
```

```
      try {
         initDBIf();
         mapper.create(counter);
      } catch (Exception ex) {
         throw new CreateException (ex.getMessage());
      }

      return counterId;
   }
```

The `ejbRemove` method is invoked to destroy a counter and remove it from the database.

```
   /**
    * Deletes the EJBBean from the persistent storage.
    */
   public void ejbRemove() {
      String id = (String) ctx.getPrimaryKey();
      System.out.println ("ejbRemove (" + id + ")");
      boolean isDeleted = true;
      try {
         initDBIf();
         isDeleted = mapper.deleteByPrimaryKey (id, CounterBean.class);
      } catch (Exception ex) {
         throw new EJBException(ex);
      }
      if (! isDeleted) {
         throw new NoSuchEntityException ("No counter with id " + id);
      }
   }
```

The following code implements all the finder methods declared in the `CounterHome` home interface. Notice that the EJB-finder methods have the same signature as the`findXXX` methods in the home interface. These finder methods are used to find existing counter beans in the database. They return either the primary key (`String` class for the counter bean) for the entity bean it finds or an enumeration of primary keys if more than one are found.

```
   /**
    * Attempts to find the EJBBean with a given Primary Key from
    * the persistent storage.
    */
   public String ejbFindByPrimaryKey (String pk)
                                 throws ObjectNotFoundException {
      System.out.println ("ejbFindByPrimaryKey (" + pk + ")");
      CounterBean bean = null;
      try {
         initDBIf();
         bean = (CounterBean) mapper.findByPrimaryKey (pk, CounterBean.class);
      } catch (Exception ex) {
         throw new EJBException (ex);
      }
      if (bean != null) {
         System.out.println ("ejbFindByPrimaryKey found counter[" + pk + "]");
         counter = bean;
      } else {
         throw new NoSuchEntityException ("No counter with id " + pk);
      }
      return pk;
   }

   public Enumeration ejbFindAll() {
      System.out.println ("ejbFindAll");
      try {
         initDBIf();
         Collection c = mapper.findAllPrimaryKeys (CounterBean.class);
         System.out.println ("The Collection is " + c);
         return Collections.enumeration (c);
      } catch (Exception ex) {
         throw new EJBException (ex);
```

```
      }
   }

   public Enumeration ejbFindNonZeroCounters() {
      System.out.println ("ejbFindNonZeroCounters");
      try {
         initDBIf();
         Collection c = mapper.findPrimaryKeysByQuery (
               "select id from demo_counter where value<>0", CounterBean.class);
         return Collections.enumeration (c);
      } catch (Exception ex) {
         throw new EJBException (ex);
      }
   }
```

Implementation of remote interface methods (see `Counter.java`);

```
   public int increment () {
      System.out.println("Incrementing counter[" + counter.getCounterId() +"]");
      return counter.increment();
   }

   public int decrement () {
      System.out.println("Decrementing counter[" + counter.getCounterId() +"]");
      return counter.decrement();
   }

   public int value() {
      return counter.getValue();
   }
```

The other EJB-required methods that the EJB container will call to manage the counter entity bean:

```
   public void ejbActivate() {
      System.out.println ("ejbActivate (" + ctx.getPrimaryKey() + ")");
   }

   public void ejbPassivate() {
   }

   public void ejbPostCreate(String counterId) {
   }
```

The following method creates and initializes the shared mapper, `mapper,` used by this entity bean. The `mapper` object handles the object-relational mapping of counter entity beans to the database. The following code demonstrates two different ways to create the `mapper` object. The first and simplest way is to load the required data source and the mappings from a mapper configuration file. Alternatively, the `mapper` object can be instantiated by directly invoking dbMapper class methods (please see the code within the comments).

```
   private void initDBIf () throws Exception {
      if (mapper != null) {
         return;
      }

      // Get the shared singelton DBModule instance
      System.out.println ("Loading DB Mapper file and creating the mapping "
                          + "context");
      DBModule dbm = DBModule.init ("/dbmapper.xml");
      mapper = dbm.createDefaultMapper ("default");

/*
      // Uncomment following if dont want to use XML files
      // Following code creates mapper using dbMapper classes directy (refer to
      // javadoc API for details)
      DBModule dbm = DBModule.init();
      ORMapEntry mapEntry = new ORMapEntry();
      mapEntry.setClassName (CounterBean.class.getName());
```

```
      mapEntry.setTableName ("demo_counter");
      ORFieldInfo[] fields = new ORFieldInfo [2];
      fields[0] = new ORFieldInfo (mapEntry, "counterId", null, null, true,
                    new ORFieldInfo.BasicTypeInfo("String", "id"));
      fields[1] = new ORFieldInfo (mapEntry, "value", null, null, false,
                    new ORFieldInfo.BasicTypeInfo("int", "value"));
      mapEntry.setFields (fields);
      ORMappingInfo mappingInfo = new ORMappingInfo();
      mappingInfo.add (mapEntry);

      com.nec.tdd.tools.dbMapper.JNDIDataSource ds =
            new com.nec.tdd.tools.dbMapper.JNDIDataSource ("jdbc/OracleCoreDS");

      mapper = new DefaultMapper (ds, mappingInfo);
*/
   }

}
```