HIBERNATE
java persistence framework

tutorialspoint
SIMPLYEASYLEARNING

www.tutorialspoint.com

About the Tutorial

Hibernate is a high-performance Object/Relational persistence and query service, which is licensed under the open source GNU Lesser General Public License (LGPL) and is free to download. Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities.

This tutorial will teach you how to use Hibernate to develop your database based web applications in simple and easy steps.

Audience

This tutorial is designed for all those Java programmers who would like to understand the Hibernate framework and its API.

Prerequisites

We assume you have a good understanding of the Java programming language. A basic understanding of relational databases, JDBC, and SQL will be very helpful in understanding this tutorial.

Copyright & Disclaimer

Table of Contents

1. HIBERNATE – ORM OVERVIEW

What is JDBC?

JDBC stands for **Java Database Connectivity**. It provides a set of Java API for accessing the relational databases from Java program. These Java APIs enables Java programs to execute SQL statements and interact with any SQL compliant database.

JDBC provides a flexible architecture to write a database independent application that can run on different platforms and interact with different DBMS without any modification.

Pros and Cons of JDBC

Pros of JDBC	Cons of JDBC
Clean and simple SQL processing	Complex if it is used in large projects
Good performance with large data	Large programming overhead
Very good for small applications	No encapsulation
Simple syntax so easy to learn	Hard to implement MVC concept. Query is DBMS specific.

Why Object Relational Mapping (ORM)?

When we work with an object-oriented system, there is a mismatch between the object model and the relational database. RDBMSs represent data in a tabular format whereas object-oriented languages, such as Java or C# represent it as an interconnected graph of objects.

Consider the following Java Class with proper constructors and associated public function:

```
public class Employee {

   private int id;

   private String first_name;

   private String last_name;

   private int salary;


   public Employee() {}

   public Employee(String fname, String lname, int salary) {

      this.first_name = fname;

      this.last_name = lname;

      this.salary = salary;
```

```
    }
    public int getId() {

        return id;

    }

    public String getFirstName() {

        return first_name;

    }

    public String getLastName() {

        return last_name;

    }

    public int getSalary() {

        return salary;

    }

}
```

Consider the above objects are to be stored and retrieved into the following RDBMS table:

```
create table EMPLOYEE (

    id INT NOT NULL auto_increment,

    first_name VARCHAR(20) default NULL,

    last_name  VARCHAR(20) default NULL,

    salary     INT  default NULL,

    PRIMARY KEY (id)

);
```

First problem, what if we need to modify the design of our database after having developed a few pages or our application? Second, loading and storing objects in a relational database exposes us to the following five mismatch problems:

Mismatch	Description
Granularity	Sometimes you will have an object model, which has more classes than the number of corresponding tables in the database.
Inheritance	RDBMSs do not define anything similar to Inheritance, which is a natural paradigm in object-oriented programming languages.
Identity	An RDBMS defines exactly one notion of 'sameness': the primary key. Java, however, defines both object identity (a==b) and object equality (a.equals(b)).
Associations	Object-oriented languages represent associations using object references whereas an RDBMS represents an association as a foreign key column.

Navigation	The ways you access objects in Java and in RDBMS are fundamentally different.

The **O**bject-**R**elational **M**apping (ORM) is the solution to handle all the above impedance mismatches.

What is ORM?

ORM stands for **O**bject-**R**elational **M**apping (ORM) is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc.

An ORM system has the following advantages over plain JDBC:

S.N.	Advantages
1	Let's business code access objects rather than DB tables.
2	Hides details of SQL queries from OO logic.
3	Based on JDBC 'under the hood.'
4	No need to deal with the database implementation.
5	Entities based on business concepts rather than database structure.
6	Transaction management and automatic key generation.
7	Fast development of application.

An ORM solution consists of the following four entities:

S.N.	Solutions
1	An API to perform basic CRUD operations on objects of persistent classes.
2	A language or API to specify queries that refer to classes and properties of classes.
3	A configurable facility for specifying mapping metadata.
4	A technique to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions.

Java ORM Frameworks

There are several persistent frameworks and ORM options in Java. A persistent framework is an ORM service that stores and retrieves objects into a relational database.

- Enterprise JavaBeans Entity Beans

- Java Data Objects

- Castor

- TopLink

- Spring DAO

- Hibernate, and many more

Hibernate is an **O**bject-**R**elational **M**apping(ORM) solution for JAVA. It is an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieves the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.

Hibernate Advantages

- Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.

- Provides simple APIs for storing and retrieving Java objects directly to and from the database.

- If there is change in the database or in any table, then you need to change the XML file properties only.

- Abstracts away the unfamiliar SQL types and provides a way to work around familiar Java Objects.

- Hibernate does not require an application server to operate.

- Manipulates Complex associations of objects of your database.

- Minimizes database access with smart fetching strategies.

- Provides simple querying of data.

Supported Databases

Hibernate supports almost all the major RDBMS. Following is a list of few of the database engines supported by Hibernate:

- HSQL Database Engine
- DB2/NT
- MySQL
- PostgreSQL
- FrontBase
- Oracle
- Microsoft SQL Server Database
- Sybase SQL Server
- Informix Dynamic Server

Supported Technologies

Hibernate supports a variety of other technologies, including:

- XDoclet Spring
- J2EE
- Eclipse plug-ins
- Maven

3. HIBERNATE ARCHITECTURE

Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

Following is a very high level view of the Hibernate Application Architecture.
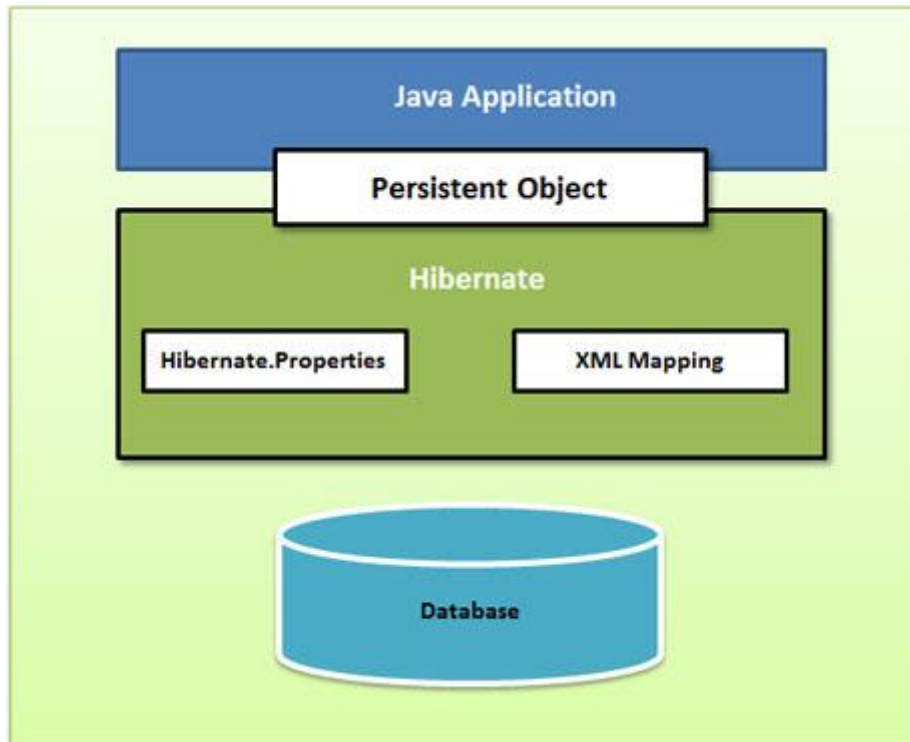
Following is a detailed view of the Hibernate Application Architecture with its important core classes.
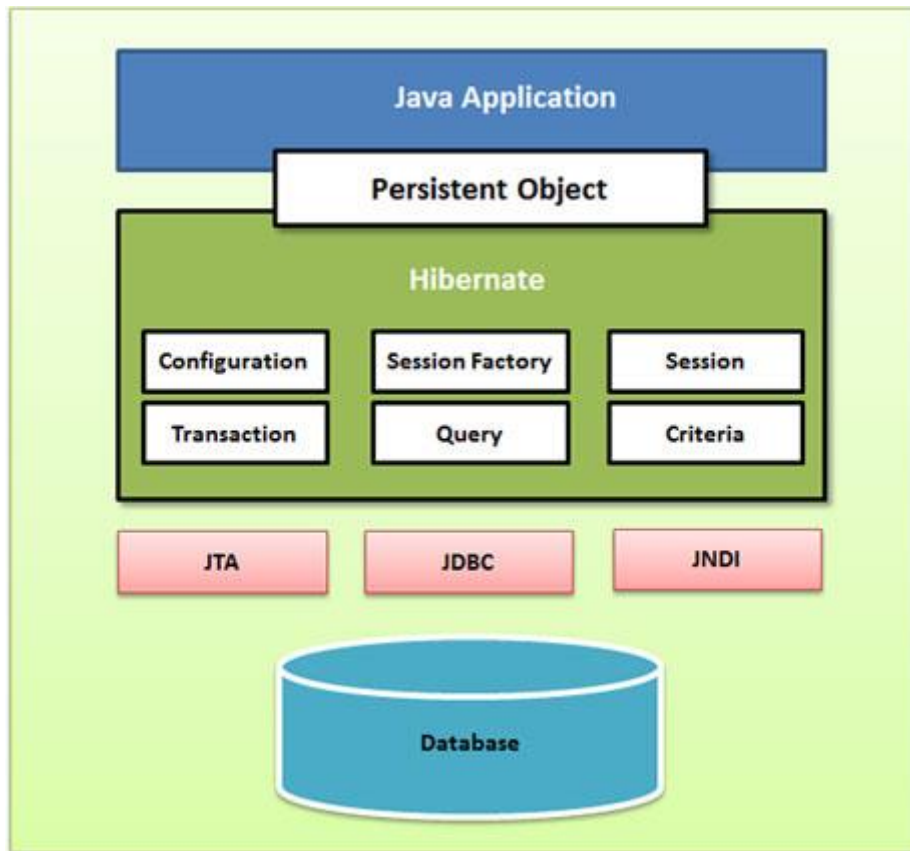
Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JTA), and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

Following section gives brief description of each of the class objects involved in Hibernate Application Architecture.

Configuration Object

The Configuration object is the first Hibernate object you create in any Hibernate application. It is usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate.

The Configuration object provides two keys components:

- **Database Connection:** This is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml**.

- **Class Mapping Setup:** This component creates the connection between the Java classes and database tables.

SessionFactory Object

Configuration object is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application.

The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use. You would need one SessionFactory object per database using a separate configuration file. So, if you are using multiple databases, then you would have to create multiple SessionFactory objects.

Session Object

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed.

Transaction Object

A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.

Query Object

Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

Criteria Object

Criteria objects are used to create and execute object oriented criteria queries to retrieve objects.
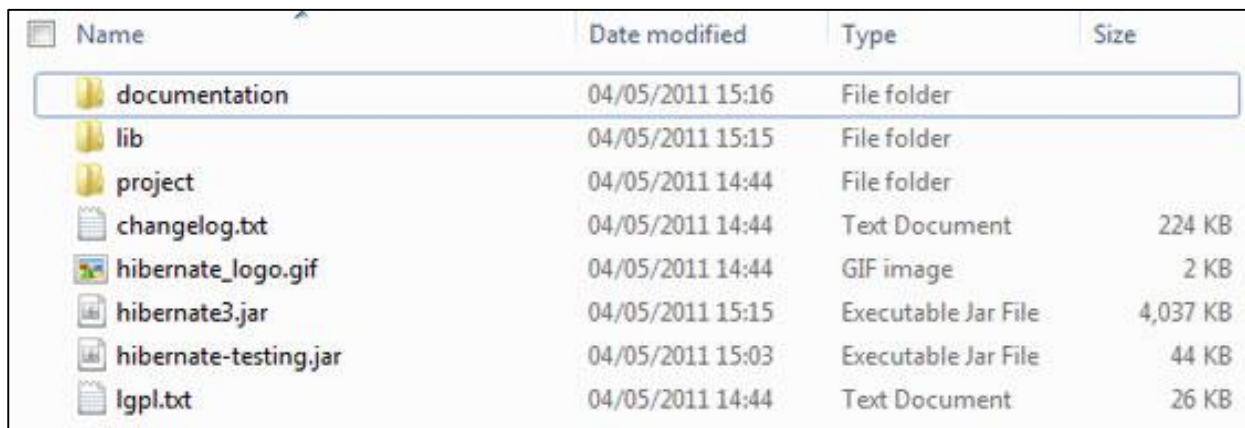
This chapter explains how to install Hibernate and other associated packages to prepare an environment for the Hibernate applications. We will work with MySQL database to experiment with Hibernate examples, so make sure you already have a setup for MySQL database. For more detail on MySQL, you can check our MySQL Tutorial.

Downloading Hibernate

It is assumed that you already have the latest version of Java installed on your system. Following are the simple steps to download and install Hibernate on your system:

- Make a choice whether you want to install Hibernate on Windows, or Unix and then proceed to the next step to download .zip file for windows and .tz file for Unix.

- Download the latest version of Hibernate from http://www.hibernate.org/downloads.

- At the time of writing this tutorial, I downloaded **hibernate-distribution-3.6.4.Final** and when you unzip the downloaded file, it will give you directory structure as shown in the following image:

Name	Date modified	Type	Size
documentation	04/05/2011 15:16	File folder	
lib	04/05/2011 15:15	File folder	
project	04/05/2011 14:44	File folder	
changelog.txt	04/05/2011 14:44	Text Document	224 KB
hibernate_logo.gif	04/05/2011 14:44	GIF image	2 KB
hibernate3.jar	04/05/2011 15:15	Executable Jar File	4,037 KB
hibernate-testing.jar	04/05/2011 15:03	Executable Jar File	44 KB
lgpl.txt	04/05/2011 14:44	Text Document	26 KB

Installing Hibernate

Once you downloaded and unzipped the latest version of the Hibernate Installation file, you need to perform following two simple steps. Make sure you are setting your CLASSPATH variable properly otherwise you will face problem while compiling your application.

- Now, copy all the library files from **/lib** into your CLASSPATH, and change your classpath variable to include all the JARs:

- Finally, copy **hibernate3.jar** file into your CLASSPATH. This file lies in the root directory of the installation and is the primary JAR that Hibernate needs to do its work.

Hibernate Prerequisites

Following is the list of the packages/libraries required by Hibernate and you should install them before starting with Hibernate. To install these packages, you will have to copy library files from **/lib** into your CLASSPATH, and change your CLASSPATH variable accordingly.

S.N.	Packages/Libraries
1	**dom4j** - XML parsing www.dom4j.org/
2	**Xalan** - XSLT Processor http://xml.apache.org/xalan-j/
3	**Xerces** - The Xerces Java Parser http://xml.apache.org/xerces-j/
4	**cglib** - Appropriate changes to Java classes at runtime http://cglib.sourceforge.net/
5	**log4j** - Logging Framework http://logging.apache.org/log4j
6	**Commons** - Logging, Email etc. http://jakarta.apache.org/commons
7	**SLF4J** - Logging Facade for Java http://www.slf4j.org

5. HIBERNATE – CONFIGURATION

Hibernate requires to know in advance — where to find the mapping information that defines how your Java classes relate to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters. All such information is usually supplied as a standard Java properties file called **hibernate.properties**, or as an XML file named **hibernate.cfg.xml**.

I will consider XML formatted file **hibernate.cfg.xml** to specify required Hibernate properties in my examples. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file is kept in the root directory of your application's classpath.

Hibernate Properties

Following is the list of important properties, you will be required to configure for a databases in a standalone situation:

S.N.	Properties and Description
1	**hibernate.dialect** This property makes Hibernate generate the appropriate SQL for the chosen database.
2	**hibernate.connection.driver_class** The JDBC driver class.
3	**hibernate.connection.url** The JDBC URL to the database instance.
4	**hibernate.connection.username** The database username.
5	**hibernate.connection.password** The database password.
6	**hibernate.connection.pool_size** Limits the number of connections waiting in the Hibernate database connection pool.
7	**hibernate.connection.autocommit** Allows auto-commit mode to be used for the JDBC connection.

If you are using a database along with an application server and JNDI, then you would have to configure the following properties:

S.N.	Properties and Description
1	**hibernate.connection.datasource** The JNDI name defined in the application server context, which you are using for the application.
2	**hibernate.jndi.class** The InitialContext class for JNDI.
3	**hibernate.jndi.<JNDIpropertyname>** Passes any JNDI property you like to the JNDI *InitialContext*.
4	**hibernate.jndi.url** Provides the URL for JNDI.
5	**hibernate.connection.username** The database username.
6	**hibernate.connection.password** The database password.

Hibernate with MySQL Database

MySQL is one of the most popular open-source database systems available today. Let us create **hibernate.cfg.xml** configuration file and place it in the root of your application's classpath. You will have to make sure that you have **testdb** database available in your MySQL database and you have a user **test** available to access the database.

The XML configuration file must conform to the Hibernate 3 Configuration DTD, which is available at http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">


<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>


   <!-- Assume test is the database name -->
```

```
    <property name="hibernate.connection.url">
        jdbc:mysql://localhost/test
    </property>
    <property name="hibernate.connection.username">
        root
    </property>
    <property name="hibernate.connection.password">
        root123
    </property>


    <!-- List of XML mapping files -->
    <mapping resource="Employee.hbm.xml"/>


</session-factory>
</hibernate-configuration>
```

The above configuration file includes **<mapping>** tags, which are related to hibernate-mapping file and we will see in next chapter what exactly a hibernate mapping file is and how and why do we use it?

Following is the list of various important databases dialect property type:

Database	Dialect Property
DB2	org.hibernate.dialect.DB2Dialect
HSQLDB	org.hibernate.dialect.HSQLDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Informix	org.hibernate.dialect.InformixDialect
Ingres	org.hibernate.dialect.IngresDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Microsoft SQL Server 2000	org.hibernate.dialect.SQLServerDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
MySQL	org.hibernate.dialect.MySQLDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 11g	org.hibernate.dialect.Oracle10gDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect

Oracle 9i	org.hibernate.dialect.Oracle9iDialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
Progress	org.hibernate.dialect.ProgressDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect

6. HIBERNATE — SESSIONS

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed. The main function of the Session is to offer, create, read, and delete operations for instances of mapped entity classes.

Instances may exist in one of the following three states at a given point in time:

- **transient:** A new instance of a persistent class, which is not associated with a Session and has no representation in the database and no identifier value is considered transient by Hibernate.

- **persistent:** You can make a transient instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.

- **detached:** Once we close the Hibernate Session, the persistent instance will become a detached instance.

A Session instance is serializable if its persistent classes are serializable. A typical transaction should use the following idiom:

```
Session session = factory.openSession();

Transaction tx = null;

try {

    tx = session.beginTransaction();

    // do some work

    ...

    tx.commit();

}

catch (Exception e) {

    if (tx!=null) tx.rollback();

    e.printStackTrace();

}finally {

    session.close();

}
```

If the Session throws an exception, the transaction must be rolled back and the session must be discarded.

Session Interface Methods

There are number of methods provided by the **Session** interface, but I'm going to list down a few important methods only, which we will use in this tutorial. You can check Hibernate documentation for a complete list of methods associated with **Session** and **SessionFactory**.

S.N.	Session Methods and Description
1	**Transaction beginTransaction()** Begin a unit of work and return the associated Transaction object.
2	**void cancelQuery()** Cancel the execution of the current query.
3	**void clear()** Completely clear the session.
4	**Connection close()** End the session by releasing the JDBC connection and cleaning up.
5	**Criteria createCriteria(Class persistentClass)** Create a new Criteria instance, for the given entity class, or a superclass of an entity class.
6	**Criteria createCriteria(String entityName)** Create a new Criteria instance, for the given entity name.
7	**Serializable getIdentifier(Object object)** Return the identifier value of the given entity as associated with this session.
8	**Query createFilter(Object collection, String queryString)** Create a new instance of Query for the given collection and filter string.
9	**Query createQuery(String queryString)** Create a new instance of Query for the given HQL query string.
10	**SQLQuery createSQLQuery(String queryString)** Create a new instance of SQLQuery for the given SQL query string.
11	**void delete(Object object)** Remove a persistent instance from the datastore.
12	**void delete(String entityName, Object object)** Remove a persistent instance from the datastore.
13	**Session get(String entityName, Serializable id)** Return the persistent instance of the given named entity with the given identifier, or null if there is no such persistent instance.
14	**SessionFactory getSessionFactory()** Get the session factory, which created this session.
15	**void refresh(Object object)** Re-read the state of the given instance from the underlying database.
16	**Transaction getTransaction()** Get the Transaction instance associated with this session.
17	**boolean isConnected()** Check if the session is currently connected.

18	**boolean isDirty()** Does this session contain any changes, which must be synchronized with the database?
19	**boolean isOpen()** Check if the session is still open.
20	**Serializable save(Object object)** Persist the given transient instance, first assigning a generated identifier.
21	**void saveOrUpdate(Object object)** Either save(Object) or update(Object) the given instance.
22	**void update(Object object)** Update the persistent instance with the identifier of the given detached instance.
23	**void update(String entityName, Object object)** Update the persistent instance with the identifier of the given detached instance.

The entire concept of Hibernate is to take the values from Java class attributes and persist them to a database table. A mapping document helps Hibernate in determining how to pull the values from the classes and map them with table and associated fields.

Java classes whose objects or instances will be stored in database tables are called persistent classes in Hibernate. Hibernate works best if these classes follow some simple rules, also known as the **Plain Old Java Object** (POJO) programming model.

There are following main rules of persistent classes, however, none of these rules are hard requirements:

- All Java classes that will be persisted need a default constructor.

- All classes should contain an ID in order to allow easy identification of your objects within Hibernate and the database. This property maps to the primary key column of a database table.

- All attributes that will be persisted should be declared private and have **getXXX** and **setXXX** methods defined in the JavaBean style.

- A central feature of Hibernate, proxies, depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods.

- All classes that do not extend or implement some specialized classes and interfaces required by the EJB framework.

The POJO name is used to emphasize that a given object is an ordinary Java Object, not a special object, and in particular not an Enterprise JavaBean.

Simple POJO Example

Based on the few rules mentioned above, we can define a POJO class as follows:

```java
public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;
public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
```

```
   }
   public int getId() {

      return id;

   }
   public void setId( int id ) {

      this.id = id;

   }
   public String getFirstName() {

      return firstName;

   }
   public void setFirstName( String first_name ) {

      this.firstName = first_name;

   }
   public String getLastName() {

      return lastName;

   }
   public void setLastName( String last_name ) {

      this.lastName = last_name;

   }
   public int getSalary() {

      return salary;

   }
   public void setSalary( int salary ) {

      this.salary = salary;

   }
}
```

An Object/relational mappings are usually defined in an XML document. This mapping file instructs Hibernate — how to map the defined class or classes to the database tables?

Though many Hibernate users choose to write the XML by hand, but a number of tools exist to generate the mapping document. These include **XDoclet, Middlegen,** and **AndroMDA** for the advanced Hibernate users.

Let us consider our previously defined POJO class whose objects will persist in the table defined in next section.

```java
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;


   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
```

```
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

There would be one table corresponding to each object you are willing to provide persistence. Consider above objects need to be stored and retrieved into the following RDBMS table:

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Based on the two above entities, we can define following mapping file, which instructs Hibernate how to map the defined class or classes to the database tables.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
```

```
            <generator class="native"/>

        </id>

        <property name="firstName" column="first_name" type="string"/>

        <property name="lastName" column="last_name" type="string"/>

        <property name="salary" column="salary" type="int"/>

    </class>

</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml.

Let us see understand a little detail about the mapping elements used in the mapping file:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains all the <class> elements.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence,** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

There are other attributes and elements available, which will be used in a mapping document and I would try to cover as many as possible while discussing other Hibernate related topics.

When you prepare a Hibernate mapping document, you find that you map the Java data types into RDBMS data types. The **types** declared and used in the mapping files are not Java data types; they are not SQL database types either. These types are called **Hibernate mapping types**, which can translate from Java to SQL data types and vice versa.

This chapter lists down all the basic, date and time, large object, and various other built-in mapping types.

Primitive Types

Mapping type	Java type	ANSI SQL Type
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte or java.lang.Byte	TINYINT
boolean	boolean or java.lang.Boolean	BIT
yes/no	boolean or java.lang.Boolean	CHAR(1) ('Y' or 'N')
true/false	boolean or java.lang.Boolean	CHAR(1) ('T' or 'F')

Date and Time Types

Mapping type	Java type	ANSI SQL Type
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

Binary and Large Object Types

Mapping type	Java type	ANSI SQL Type

binary	byte[]	VARBINARY (or BLOB)
text	java.lang.String	CLOB
serializable	any Java class that implements java.io.Serializable	VARBINARY (or BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

JDK-related Types

Mapping type	Java type	ANSI SQL Type
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

Let us now take an example to understand how we can use Hibernate to provide Java persistence in a standalone application. We will go through the different steps involved in creating a Java Application using Hibernate technology.

Create POJO Classes

The first step in creating an application is to build the Java POJO class or classes, depending on the application that will be persisted to the database. Let us consider our **Employee** class with **getXXX** and **setXXX** methods to make it JavaBeans compliant class.

A POJO (Plain Old Java Object) is a Java object that doesn't extend or implement some specialized classes and interfaces respectively required by the EJB framework. All normal Java objects are POJO.

When you design a class to be persisted by Hibernate, it is important to provide JavaBeans compliant code as well as one attribute, which would work as index like **id** attribute in the Employee class.

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
```

```
      return firstName;
   }
   public void setFirstName( String first_name ) {

      this.firstName = first_name;

   }
   public String getLastName() {

      return lastName;

   }
   public void setLastName( String last_name ) {

      this.lastName = last_name;

   }
   public int getSalary() {

      return salary;

   }
   public void setSalary( int salary ) {

      this.salary = salary;

   }
}
```

Create Database Tables

Second step would be creating tables in your database. There would be one table corresponding to each object, you are willing to provide persistence. Consider above objects need to be stored and retrieved into the following RDBMS table:

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Create Mapping Configuration File

This step is to create a mapping file that instructs Hibernate how to map the defined class or classes to the database tables.

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>
</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. Let us see little detail about the mapping document:

- The mapping document is an XML document having <hibernate-mapping> as the root element, which contains all the <class> elements.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence,** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The

tutorialspoint
SIMPLYEASYLEARNING

type attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

There are other attributes and elements available, which will be used in a mapping document and I would try to cover as many as possible while discussing other Hibernate related topics.

Create Application Class

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employee's records and then we will apply CRUD operations on those records.

```java
import java.util.List;

import java.util.Date;

import java.util.Iterator;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {

    private static SessionFactory factory;

    public static void main(String[] args) {

        try{

            factory = new Configuration().configure().buildSessionFactory();

        }catch (Throwable ex) {

            System.err.println("Failed to create sessionFactory object." + ex);

            throw new ExceptionInInitializerError(ex);

        }

        ManageEmployee ME = new ManageEmployee();


        /* Add few employee records in database */

        Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);

        Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);

        Integer empID3 = ME.addEmployee("John", "Paul", 10000);


        /* List down all the employees */
```

```
   ME.listEmployees();


   /* Update employee's records */

   ME.updateEmployee(empID1, 5000);


   /* Delete an employee from the database */

   ME.deleteEmployee(empID2);


   /* List down new list of the employees */

   ME.listEmployees();

}
/* Method to CREATE an employee in the database */
public Integer addEmployee(String fname, String lname, int salary){
   Session session = factory.openSession();
   Transaction tx = null;
   Integer employeeID = null;
   try{
      tx = session.beginTransaction();
      Employee employee = new Employee(fname, lname, salary);
      employeeID = (Integer) session.save(employee);
      tx.commit();
   }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   }finally {
      session.close();
   }
   return employeeID;
}
/* Method to  READ all the employees */
public void listEmployees( ){
   Session session = factory.openSession();
   Transaction tx = null;
   try{
      tx = session.beginTransaction();
```

```
        List employees = session.createQuery("FROM Employee").list();
        for (Iterator iterator =
                        employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
            System.out.println("  Salary: " + employee.getSalary());
        }
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}
/* Method to UPDATE salary for an employee */
public void updateEmployee(Integer EmployeeID, int salary ){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        Employee employee =
                (Employee)session.get(Employee.class, EmployeeID);
        employee.setSalary( salary );
            session.update(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}
/* Method to DELETE an employee from the records */
```

```
   public void deleteEmployee(Integer EmployeeID){

      Session session = factory.openSession();

      Transaction tx = null;

      try{

         tx = session.beginTransaction();

         Employee employee =

                  (Employee)session.get(Employee.class, EmployeeID);

         session.delete(employee);

         tx.commit();

      }catch (HibernateException e) {

         if (tx!=null) tx.rollback();

         e.printStackTrace();

      }finally {

         session.close();

      }

   }

}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure, you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result, and records would be created in the EMPLOYEE table.

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........

First Name: Zara   Last Name: Ali   Salary: 1000

First Name: Daisy  Last Name: Das   Salary: 5000

First Name: John   Last Name: Paul  Salary: 10000

First Name: Zara   Last Name: Ali   Salary: 5000
```

```
First Name: John   Last Name: Paul   Salary: 10000
```

If you check your EMPLOYEE table, it should have the following records:

```
mysql> select * from EMPLOYEE;

+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
| 29 | Zara       | Ali       |   5000 |
| 31 | John       | Paul      |  10000 |
+----+------------+-----------+--------+
2 rows in set (0.00 sec


mysql>
```

So far, we have seen very basic O/R mapping using hibernate, but there are three most important mapping topics, which we have to learn in detail.

These are:

- Mapping of collections,

- Mapping of associations between entity classes, and

- Component Mappings.

Collections Mappings

If an entity or class has collection of values for a particular variable, then we can map those values using any one of the collection interfaces available in java. Hibernate can persist instances of **java.util.Map, java.util.Set, java.util.SortedMap, java.util.SortedSet, java.util.List**, and any **array** of persistent entities or values.

Collection type	Mapping and Description
java.util.Set	This is mapped with a <set> element and initialized with java.util.HashSet
java.util.SortedSet	This is mapped with a <set> element and initialized with java.util.TreeSet. The **sort** attribute can be set to either a comparator or natural ordering.
java.util.List	This is mapped with a <list> element and initialized with java.util.ArrayList
java.util.Collection	This is mapped with a <bag> or <ibag> element and initialized with java.util.ArrayList
java.util.Map	This is mapped with a <map> element and initialized with java.util.HashMap
java.util.SortedMap	This is mapped with a <map> element and initialized with java.util.TreeMap. The **sort** attribute can be set to either a comparator or natural ordering.

Arrays are supported by Hibernate with <primitive-array> for Java primitive value types and <array> for everything else. However, they are rarely used, so I am not going to discuss them in this tutorial.

If you want to map a user defined collection interfaces, which is not directly supported by Hibernate, you need to tell Hibernate about the semantics of your custom collections, which is not very easy and not recommend to be used.

Hibernate – Set Mappings

A **Set** is a java collection that does not contain any duplicate element. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most, one null element. So, objects to be added to a set must implement both the equals() and hashCode() methods so that Java can determine whether any two elements/objects are identical.

A Set is mapped with a <set> element in the mapping table and initialized with java.util.HashSet. You can use Set collection in your class when there is no duplicate element required in the collection.

Define RDBMS Tables

Consider a situation where we need to store our employee records in EMPLOYEE table, which would have the following structure:

```
create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name  VARCHAR(20) default NULL,
    salary     INT  default NULL,
    PRIMARY KEY (id)
);
```

Further, assume each employee can have one or more certificate associated with him/her. So, we will store certificate related information in a separate table having the following structure:

```
create table CERTIFICATE (
    id INT NOT NULL auto_increment,
    certificate_name VARCHAR(30) default NULL,
    employee_id INT default NULL,
    PRIMARY KEY (id)
);
```

There will be **one-to-many** relationship between EMPLOYEE and CERTIFICATE objects:

Define POJO Classes

Let us implement our POJO class **Employee,** which will be used to persist the objects related to EMPLOYEE table and having a collection of certificates in **Set** variable.

```
import java.util.*;

public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;
   private Set certificates;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
```

```
   }
   public void setSalary( int salary ) {

      this.salary = salary;

   }


   public Set getCertificates() {

      return certificates;

   }
   public void setCertificates( Set certificates ) {

      this.certificates = certificates;

   }

}
```

Now let us define another POJO class corresponding to CERTIFICATE table so that certificate objects can be stored and retrieved into the CERTIFICATE table. This class should also implement both the equals() and hashCode() methods so that Java can determine whether any two elements/objects are identical.

```
public class Certificate {

   private int id;

   private String name;

   public Certificate() {}

   public Certificate(String name) {

      this.name = name;

   }

   public int getId() {

      return id;

   }

   public void setId( int id ) {

      this.id = id;

   }

   public String getName() {

      return name;

   }

   public void setName( String name ) {

      this.name = name;

   }
```

```java
    public boolean equals(Object obj) {
        if (obj == null) return false;
        if (!this.getClass().equals(obj.getClass())) return false;

        Certificate obj2 = (Certificate)obj;
        if((this.id == obj2.getId()) && (this.name.equals(obj2.getName())))
        {
            return true;
        }
        return false;
    }
    public int hashCode() {
        int tmp = 0;
        tmp = ( id + name ).hashCode();
        return tmp;
    }
}
```

Define Hibernate Mapping File

Let us develop our mapping file, which instructs Hibernate how to map the defined classes to the database tables. The <set> element will be used to define the rule for Set collection used.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
```

```
        <set name="certificates" cascade="all">

            <key column="employee_id"/>

            <one-to-many class="Certificate"/>

        </set>

        <property name="firstName" column="first_name" type="string"/>

        <property name="lastName" column="last_name" type="string"/>

        <property name="salary" column="salary" type="int"/>

    </class>


    <class name="Certificate" table="CERTIFICATE">

        <meta attribute="class-description">

            This class contains the certificate records.

        </meta>

        <id name="id" type="int" column="id">

            <generator class="native"/>

        </id>

        <property name="name" column="certificate_name" type="string"/>

    </class>


</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. You are already familiar with most of the mapping detail, but let us see all the elements of mapping file once again:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains two <class> elements corresponding to each class.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to

native to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<set>** element is new here and has been introduced to set the relationship between Certificate and Employee classes. We used the **cascade** attribute in the <set> element to tell Hibernate to persist the Certificate objects at the same time as the Employee objects. The **name** attribute is set to the defined **Set** variable in the parent class, in our case, it is *certificates*. For each set variable, we need to define a separate set element in the mapping file.

- The **<key>** element is the column in the CERTIFICATE table that holds the foreign key to the parent object i.e. table EMPLOYEE.

- The **<one-to-many>** element indicates that one Employee object relates to many Certificate objects and, as such, the Certificate object must have an Employee parent associated with it. You can use either **<one-to-one>**, **<many-to-one>** or **<many-to-many>** elements based on your requirement.

Create Application Class

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employees' records along with their certificates and then we will apply CRUD operations on those records.

```
import java.util.*;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {

   private static SessionFactory factory;

   public static void main(String[] args) {

      try{

         factory = new Configuration().configure().buildSessionFactory();

      }catch (Throwable ex) {

         System.err.println("Failed to create sessionFactory object." + ex);

         throw new ExceptionInInitializerError(ex);
```

```
      }
      ManageEmployee ME = new ManageEmployee();

      /* Let us have a set of certificates for the first employee  */
      HashSet set1 = new HashSet();
      set1.add(new Certificate("MCA"));
      set1.add(new Certificate("MBA"));
      set1.add(new Certificate("PMP"));


      /* Add employee records in the database */
      Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, set1);


      /* Another set of certificates for the second employee  */
      HashSet set2 = new HashSet();
      set2.add(new Certificate("BCA"));
      set2.add(new Certificate("BA"));


      /* Add another employee record in the database */
      Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, set2);


      /* List down all the employees */
      ME.listEmployees();


      /* Update employee's salary records */
      ME.updateEmployee(empID1, 5000);


      /* Delete an employee from the database */
      ME.deleteEmployee(empID2);


      /* List down all the employees */
      ME.listEmployees();

   }


   /* Method to add an employee record in the database */
   public Integer addEmployee(String fname, String lname,
```

```
                                        int salary, Set cert){
    Session session = factory.openSession();

    Transaction tx = null;

    Integer employeeID = null;

    try{

        tx = session.beginTransaction();

        Employee employee = new Employee(fname, lname, salary);

        employee.setCertificates(cert);

        employeeID = (Integer) session.save(employee);

        tx.commit();

    }catch (HibernateException e) {

        if (tx!=null) tx.rollback();

        e.printStackTrace();

    }finally {

        session.close();

    }

    return employeeID;

}


/* Method to list all the employees detail */

public void listEmployees( ){

    Session session = factory.openSession();

    Transaction tx = null;

    try{

        tx = session.beginTransaction();

        List employees = session.createQuery("FROM Employee").list();

        for (Iterator iterator1 =
                        employees.iterator(); iterator1.hasNext();){

            Employee employee = (Employee) iterator1.next();

            System.out.print("First Name: " + employee.getFirstName());

            System.out.print("  Last Name: " + employee.getLastName());

            System.out.println("  Salary: " + employee.getSalary());

            Set certificates = employee.getCertificates();

            for (Iterator iterator2 =
                        certificates.iterator(); iterator2.hasNext();){
```

```
                    Certificate certName = (Certificate) iterator2.next();

                    System.out.println("Certificate: " + certName.getName());

            }

        }

        tx.commit();

    }catch (HibernateException e) {

        if (tx!=null) tx.rollback();

        e.printStackTrace();

    }finally {

        session.close();

    }

}

/* Method to update salary for an employee */

public void updateEmployee(Integer EmployeeID, int salary ){

    Session session = factory.openSession();

    Transaction tx = null;

    try{

        tx = session.beginTransaction();

        Employee employee =

                    (Employee)session.get(Employee.class, EmployeeID);

        employee.setSalary( salary );

        session.update(employee);

        tx.commit();

    }catch (HibernateException e) {

        if (tx!=null) tx.rollback();

        e.printStackTrace();

    }finally {

        session.close();

    }

}

/* Method to delete an employee from the records */

public void deleteEmployee(Integer EmployeeID){

    Session session = factory.openSession();

    Transaction tx = null;

    try{
```

```
        tx = session.beginTransaction();
        Employee employee =
                    (Employee)session.get(Employee.class, EmployeeID);
        session.delete(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
  }
}
```

Compilation and Execution

Here are the steps to compile and run the above application. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create Certificate.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result on the screen, and at the same time, records would be created in EMPLOYEE and CERTIFICATE tables.

```
$java ManageEmployee
.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Manoj   Last Name: Kumar   Salary: 4000
Certificate: MBA
Certificate: PMP
Certificate: MCA
First Name: Dilip   Last Name: Kumar   Salary: 3000
Certificate: BCA
```

```
Certificate: BA

First Name: Manoj  Last Name: Kumar  Salary: 5000

Certificate: MBA

Certificate: PMP

Certificate: MCA
```

If you check your EMPLOYEE and CERTIFICATE tables, they should have the following records:

```
mysql> select * from employee;

+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
| 1  | Manoj      | Kumar     |   5000 |
+----+------------+-----------+--------+
1 row in set (0.00 sec)


mysql> select * from certificate;

+----+------------------+-------------+
| id | certificate_name | employee_id |
+----+------------------+-------------+
| 1  | MBA              |           1 |
| 2  | PMP              |           1 |
| 3  | MCA              |           1 |
+----+------------------+-------------+
3 rows in set (0.00 sec)


mysql>
```

Hibernate – SortedSet Mappings

A **SortedSet** is a java collection that does not contain any duplicate element and elements are ordered using their natural ordering or by a comparator provided.

A SortedSet is mapped with a <set> element in the mapping table and initialized with java.util.TreeSet. The **sort** attribute can be set to either a comparator or natural ordering. If we use **natural** ordering, then its iterator will traverse the set in ascending element order.

Define RDBMS Tables

Consider a situation where we need to store our employee records in the EMPLOYEE table, which will have the following structure:

```
create table EMPLOYEE (

   id INT NOT NULL auto_increment,

   first_name VARCHAR(20) default NULL,

   last_name  VARCHAR(20) default NULL,

   salary     INT  default NULL,

   PRIMARY KEY (id)

);
```

Further, assume each employee can have one or more certificate associated with him/her. So, we will store certificate related information in a separate table having the following structure:

```
create table CERTIFICATE (

   id INT NOT NULL auto_increment,

   certificate_name VARCHAR(30) default NULL,

   employee_id INT default NULL,

   PRIMARY KEY (id)

);
```

There will be **one-to-many** relationship between EMPLOYEE and CERTIFICATE objects:

Define POJO Classes

Let us implement our POJO class **Employee,** which will be used to persist the objects related to EMPLOYEE table and having a collection of certificates in **SortedSet** variable.

```
import java.util.*;


public class Employee {

   private int id;

   private String firstName;

   private String lastName;

   private int salary;

   private SortedSet certificates;


   public Employee() {}

   public Employee(String fname, String lname, int salary) {
```

```
      this.firstName = fname;

      this.lastName = lname;

      this.salary = salary;

   }

   public int getId() {

      return id;

   }

   public void setId( int id ) {

      this.id = id;

   }

   public String getFirstName() {

      return firstName;

   }

   public void setFirstName( String first_name ) {

      this.firstName = first_name;

   }

   public String getLastName() {

      return lastName;

   }

   public void setLastName( String last_name ) {

      this.lastName = last_name;

   }

   public int getSalary() {

      return salary;

   }

   public void setSalary( int salary ) {

      this.salary = salary;

   }

   public SortedSet getCertificates() {

      return certificates;

   }

   public void setCertificates( SortedSet certificates ) {

      this.certificates = certificates;

   }

}
```

Now let us define another POJO class corresponding to CERTIFICATE table so that certificate objects can be stored and retrieved into the CERTIFICATE table. This class should also implement **Comparable** interface and **compareTo** method, which will be used to sort the elements in case you set **sort="natural"** in your mapping file (see below mapping file):

```
public class Certificate implements Comparable <Certificate>{
    private int id;
    private String name;

    public Certificate() {}
    public Certificate(String name) {
        this.name = name;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName( String name ) {
        this.name = name;
    }
    public int compareTo(Certificate that){
        final int BEFORE = -1;
        final int AFTER = 1;

        if (that == null) {
            return BEFORE;
        }

        Comparable thisCertificate = this.getName();
        Comparable thatCertificate = that.getName();
```

```
        if(thisCertificate == null) {

            return AFTER;

        } else if(thatCertificate == null) {

            return BEFORE;

        } else {

            return thisCertificate.compareTo(thatCertificate);

        }

    }

}
```

Define Hibernate Mapping File

Let us develop our mapping file, which instructs Hibernate how to map the defined classes to the database tables. The <set> element will be used to define the rule for SortedSet collection used.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <set name="certificates" cascade="all" sort="MyClass">
         <key column="employee_id"/>
         <one-to-many class="Certificate"/>
      </set>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
```

```
    </class>


    <class name="Certificate" table="CERTIFICATE">
       <meta attribute="class-description">
          This class contains the certificate records.
       </meta>
       <id name="id" type="int" column="id">
          <generator class="native"/>
       </id>
       <property name="name" column="certificate_name" type="string"/>
    </class>


</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. You are already familiar with most of the mapping detail, but let us see all the elements of mapping file once again:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains two <class> elements corresponding to each class.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<set>** element is used to set the relationship between Certificate and Employee classes. We used the **cascade** attribute in the <set> element to tell

Hibernate to persist the Certificate objects at the same time as the Employee objects. The **name** attribute is set to the defined **SortedSet** variable in the parent class, in our case it is *certificates*. The **sort** attribute can be set to **natural** to have natural sorting or it can be set to a custom class implementing **java.util.Comparator**. We have used a class **MyClass,** which implements java.util.Comparator to reverse the sorting order implemented in **Certificate** class.

- The **<key>** element is the column in the CERTIFICATE table that holds the foreign key to the parent object i.e. table EMPLOYEE.

- The **<one-to-many>** element indicates that one Employee object relates to many Certificate objects and, as such, the Certificate object must have an Employee parent associated with it. You can use either **<one-to-one>**, **<many-to-one>** or **<many-to-many>** elements based on your requirement.

If we use **sort="natural"** setting, then we do not need to create a separate class because Certificate class already has implemented Comparable interface and hibernate will use compareTo() method defined in Certificate class to compare certificate names. But we are using a custom comparator class **MyClass** in our mapping file so we would have to create this class based on our sorting algorithm. Let us do descending sorting in this class using this class.

```java
import java.util.Comparator;

public class MyClass implements Comparator<Certificate>{
   public int compare(Certificate o1, Certificate o2) {
      final int BEFORE = -1;
      final int AFTER = 1;

      /* To reverse the sorting order, multiple by -1 */
      if (o2 == null) {
         return BEFORE * -1;
      }

      Comparable thisCertificate = o1.getName();
      Comparable thatCertificate = o2.getName();

      if(thisCertificate == null) {
         return AFTER * 1;
      } else if(thatCertificate == null) {
         return BEFORE * -1;
      } else {
         return thisCertificate.compareTo(thatCertificate) * -1;
```

```
        }
    }
}
```

Create Application Class

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employees' records along with their certificates and then we will apply CRUD operations on those records.

```java
import java.util.*;

import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {
        try{
            factory = new Configuration().configure().buildSessionFactory();
        }catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
        ManageEmployee ME = new ManageEmployee();
        /* Let us have a set of certificates for the first employee  */
        TreeSet set1 = new TreeSet();
        set1.add(new Certificate("MCA"));
        set1.add(new Certificate("MBA"));
        set1.add(new Certificate("PMP"));

        /* Add employee records in the database */
        Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, set1);
```

```
   /* Another set of certificates for the second employee  */
   TreeSet set2 = new TreeSet();
   set2.add(new Certificate("BCA"));
   set2.add(new Certificate("BA"));


   /* Add another employee record in the database */
   Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, set2);


   /* List down all the employees */
   ME.listEmployees();


   /* Update employee's salary records */
   ME.updateEmployee(empID1, 5000);


   /* Delete an employee from the database */
   ME.deleteEmployee(empID2);


   /* List down all the employees */
   ME.listEmployees();

}


/* Method to add an employee record in the database */
public Integer addEmployee(String fname, String lname,
                                int salary, SortedSet cert){
   Session session = factory.openSession();
   Transaction tx = null;
   Integer employeeID = null;
   try{
      tx = session.beginTransaction();
      Employee employee = new Employee(fname, lname, salary);
      employee.setCertificates(cert);
      employeeID = (Integer) session.save(employee);
      tx.commit();
   }catch (HibernateException e) {
```

```
        if (tx!=null) tx.rollback();

        e.printStackTrace();

    }finally {

        session.close();

    }

    return employeeID;

}


/* Method to list all the employees detail */

public void listEmployees( ){

    Session session = factory.openSession();

    Transaction tx = null;

    try{

        tx = session.beginTransaction();

        List employees = session.createQuery("FROM Employee").list();

        for (Iterator iterator1 =

                        employees.iterator(); iterator1.hasNext();){

            Employee employee = (Employee) iterator1.next();

            System.out.print("First Name: " + employee.getFirstName());

            System.out.print("  Last Name: " + employee.getLastName());

            System.out.println("  Salary: " + employee.getSalary());

            SortedSet certificates = employee.getCertificates();

            for (Iterator iterator2 =

                        certificates.iterator(); iterator2.hasNext();){

                Certificate certName = (Certificate) iterator2.next();

                System.out.println("Certificate: " + certName.getName());

            }

        }

        tx.commit();

    }catch (HibernateException e) {

        if (tx!=null) tx.rollback();

        e.printStackTrace();

    }finally {

        session.close();

    }
```

```
   }
   /* Method to update salary for an employee */
   public void updateEmployee(Integer EmployeeID, int salary ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                   (Employee)session.get(Employee.class, EmployeeID);
         employee.setSalary( salary );
         session.update(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to delete an employee from the records */
   public void deleteEmployee(Integer EmployeeID){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                   (Employee)session.get(Employee.class, EmployeeID);
         session.delete(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
```

```
    }
}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create Certificate.java source file as shown above and compile it.

- Create MyClass.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result on the screen, and same time records would be created in EMPLOYEE and CERTIFICATE tables. You can see certificates have been sorted in reverse order. You can try by changing your mapping file, simply set **sort="natural"** and execute your program and compare the results.

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Manoj  Last Name: Kumar  Salary: 4000

Certificate: PMP

Certificate: MCA

Certificate: MBA

First Name: Dilip  Last Name: Kumar  Salary: 3000

Certificate: BCA

Certificate: BA

First Name: Manoj  Last Name: Kumar  Salary: 5000

Certificate: PMP

Certificate: MCA

Certificate: MBA
```

If you check your EMPLOYEE and CERTIFICATE tables, they should have the following records:

```
mysql> select * from employee;
```

```
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
| 1  | Manoj      | Kumar     |   5000 |
+----+------------+-----------+--------+
1 row in set (0.00 sec)


mysql> select * from certificate;
+----+------------------+-------------+
| id | certificate_name | employee_id |
+----+------------------+-------------+
| 1  | MBA              |           1 |
| 2  | PMP              |           1 |
| 3  | MCA              |           1 |
+----+------------------+-------------+
3 rows in set (0.00 sec)


mysql>
```

Hibernate – List Mappings

A **List** is a java collection that stores elements in sequence and allow duplicate elements. The user of this interface has precise control over where in the list, each element is inserted. The user can access elements by their integer index, and search for elements in the list. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all.

A List is mapped with a <list> element in the mapping table and initialized with java.util.ArrayList.

Define RDBMS Tables

Consider a situation where we need to store our employee records in EMPLOYEE table, which will have the following structure:

```
create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name  VARCHAR(20) default NULL,
```

```
    salary      INT  default NULL,
    PRIMARY KEY (id)
);
```

Further, assume each employee can have one or more certificate associated with him/her. A List collection mapping needs an index column in the collection table. The index column defines the position of the element in the collection. So, we will store certificate related information in a separate table having the following structure:

```
create table CERTIFICATE (
    id INT NOT NULL auto_increment,
    certificate_name VARCHAR(30) default NULL,
    idx INT default NULL,
    employee_id INT default NULL,
    PRIMARY KEY (id)
);
```

There will be **one-to-many** relationship between EMPLOYEE and CERTIFICATE objects.

Define POJO Classes

Let us implement a POJO class **Employee,** which will be used to persist the objects related to EMPLOYEE table and having a collection of certificates in **List** variable.

```
import java.util.*;

public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;
    private List certificates;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
    }
    public int getId() {
```

```
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
      this.salary = salary;
   }


   public List getCertificates() {
      return certificates;
   }
   public void setCertificates( List certificates ) {
      this.certificates = certificates;
   }
}
```

We need to define another POJO class corresponding to CERTIFICATE table so that certificate objects can be stored and retrieved into the CERTIFICATE table.

```
public class Certificate{
```

```
    private int id;
    private String name;


    public Certificate() {}
    public Certificate(String name) {
        this.name = name;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName( String name ) {
        this.name = name;
    }
}
```

Define Hibernate Mapping File

Let us develop our mapping file, which instructs Hibernate how to map the defined classes to the database tables. The <list> element will be used to define the rule for List collection used. The index of list is always of type integer and is mapped using the <list-index> element.

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>
    <class name="Employee" table="EMPLOYEE">
        <meta attribute="class-description">
            This class contains the employee detail.
        </meta>
```

```
        <id name="id" type="int" column="id">

            <generator class="native"/>

        </id>

        <list name="certificates" cascade="all">

            <key column="employee_id"/>

            <list-index column="idx"/>

            <one-to-many class="Certificate"/>

        </list>

        <property name="firstName" column="first_name" type="string"/>

        <property name="lastName" column="last_name" type="string"/>

        <property name="salary" column="salary" type="int"/>

    </class>


    <class name="Certificate" table="CERTIFICATE">

        <meta attribute="class-description">

            This class contains the certificate records.

        </meta>

        <id name="id" type="int" column="id">

            <generator class="native"/>

        </id>

        <property name="name" column="certificate_name" type="string"/>

    </class>


</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. You are already familiar with most of the mapping detail, but let us see all the elements of mapping file once again:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains two <class> elements corresponding to each class.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the

class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<list>** element is used to set the relationship between Certificate and Employee classes. We used the **cascade** attribute in the <list> element to tell Hibernate to persist the Certificate objects at the same time as the Employee objects. The **name** attribute is set to the defined **List** variable in the parent class, in our case, it is *certificates*.

- The **<key>** element is the column in the CERTIFICATE table that holds the foreign key to the parent object i.e. table EMPLOYEE.

- The **<list-index>** element is used to keep the position of the element and map with the index column in the collection table. The index of the persistent list starts at zero. You could change this, for example, with <list-index base="1".../> in your mapping.

- The **<one-to-many>** element indicates that one Employee object relates to many Certificate objects and, as such, the Certificate object must have an Employee parent associated with it. You can use either **<one-to-one>**, **<many-to-one>** or **<many-to-many>** elements based on your requirement. If we changed this example to use a many-to-many relationship, we would need an association table to map between the parent and the child objects.

Create Application Class

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employee's records along with their certificates and then we will apply CRUD operations on those records.

```java
import java.util.*;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

```

```
public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {
        try{
            factory = new Configuration().configure().buildSessionFactory();
        }catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
        ManageEmployee ME = new ManageEmployee();
        /* Let us have a set of certificates for the first employee  */
        ArrayList set1 = new ArrayList();
        set1.add(new Certificate("MCA"));
        set1.add(new Certificate("MBA"));
        set1.add(new Certificate("PMP"));

        /* Add employee records in the database */
        Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, set1);

        /* Another set of certificates for the second employee  */
        ArrayList set2 = new ArrayList();
        set2.add(new Certificate("BCA"));
        set2.add(new Certificate("BA"));

        /* Add another employee record in the database */
        Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, set2);

        /* List down all the employees */
        ME.listEmployees();

        /* Update employee's salary records */
        ME.updateEmployee(empID1, 5000);

        /* Delete an employee from the database */
        ME.deleteEmployee(empID2);
```

```
   /* List down all the employees */

   ME.listEmployees();


}


/* Method to add an employee record in the database */
public Integer addEmployee(String fname, String lname,
                                 int salary, ArrayList cert){
   Session session = factory.openSession();
   Transaction tx = null;
   Integer employeeID = null;
   try{
      tx = session.beginTransaction();
      Employee employee = new Employee(fname, lname, salary);
      employee.setCertificates(cert);
      employeeID = (Integer) session.save(employee);
      tx.commit();
   }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   }finally {
      session.close();
   }
   return employeeID;
}


/* Method to list all the employees detail */
public void listEmployees( ){
   Session session = factory.openSession();
   Transaction tx = null;
   try{
      tx = session.beginTransaction();
      List employees = session.createQuery("FROM Employee").list();
      for (Iterator iterator1 =
```

```
                              employees.iterator(); iterator1.hasNext();){
            Employee employee = (Employee) iterator1.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
            System.out.println("  Salary: " + employee.getSalary());
            List certificates = employee.getCertificates();
            for (Iterator iterator2 =
                        certificates.iterator(); iterator2.hasNext();){
                Certificate certName = (Certificate) iterator2.next();
                System.out.println("Certificate: " + certName.getName());
            }
         }
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to update salary for an employee */
   public void updateEmployee(Integer EmployeeID, int salary ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                    (Employee)session.get(Employee.class, EmployeeID);
         employee.setSalary( salary );
         session.update(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
```

```
        session.close();
      }

   }

   /* Method to delete an employee from the records */

   public void deleteEmployee(Integer EmployeeID){

      Session session = factory.openSession();

      Transaction tx = null;

      try{

         tx = session.beginTransaction();

         Employee employee =

                  (Employee)session.get(Employee.class, EmployeeID);

         session.delete(employee);

         tx.commit();

      }catch (HibernateException e) {

         if (tx!=null) tx.rollback();

         e.printStackTrace();

      }finally {

         session.close();

      }

   }

}
```

Compilation and Execution

Here are the steps to compile and run the above application. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create Certificate.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result on the screen, and same time records would be created in EMPLOYEE and CERTIFICATE tables. You can see, certificates have been sorted in reverse order. You can try by changing your mapping file, simply set **sort="natural"** and execute your program and compare the results.

```
$java ManageEmployee
.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Manoj  Last Name: Kumar  Salary: 4000
Certificate: MCA
Certificate: MBA
Certificate: PMP
First Name: Dilip  Last Name: Kumar  Salary: 3000
Certificate: BCA
Certificate: BA
First Name: Manoj  Last Name: Kumar  Salary: 5000
Certificate: MCA
Certificate: MBA
Certificate: PMP
```

If you check your EMPLOYEE and CERTIFICATE tables, they should have the following records:

```
mysql> select * from EMPLOYEE;
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
| 51 | Manoj      | Kumar     |   5000 |
+----+------------+-----------+--------+
1 row in set (0.00 sec)


mysql> select * from CERTIFICATE;
+----+------------------+------+-------------+
| id | certificate_name | idx  | employee_id |
+----+------------------+------+-------------+
|  6 | MCA              |    0 |          51 |
|  7 | MBA              |    1 |          51 |
|  8 | PMP              |    2 |          51 |
+----+------------------+------+-------------+
3 rows in set (0.00 sec


mysql>
```

Alternatively, you could map a Java array instead of a list. An array mapping is virtually identical to the previous example, except with different element and attribute names (<array> and <array-index>). However, for reasons explained earlier, Hibernate applications rarely use arrays.

Hibernate – Bag Mappings

A **Bag** is a java collection that stores elements without caring about the sequencing, but allow duplicate elements in the list. A bag is a random grouping of the objects in the list.

A Collection is mapped with a <bag> element in the mapping table and initialized with java.util.ArrayList.

Define RDBMS Tables

Consider a situation where we need to store our employee records in EMPLOYEE table, which will have the following structure:

```
create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name  VARCHAR(20) default NULL,
    salary     INT  default NULL,
    PRIMARY KEY (id)
);
```

Further, assume each employee can have one or more certificate associated with him/her. We will store certificate related information in a separate table having the following structure:

```
create table CERTIFICATE (
    id INT NOT NULL auto_increment,
    certificate_name VARCHAR(30) default NULL,
    employee_id INT default NULL,
    PRIMARY KEY (id)
);
```

There will be **one-to-many** relationship between EMPLOYEE and CERTIFICATE objects.

Define POJO Classes

Let us implement a POJO class **Employee,** which will be used to persist the objects related to EMPLOYEE table and having a collection of certificates in **List** variable.

```
import java.util.*;
```

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;
   private Collection certificates;


   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
```

```
      this.salary = salary;

   }


   public Collection getCertificates() {

      return certificates;

   }

   public void setCertificates( Collection certificates ) {

      this.certificates = certificates;

   }

}
```

We need to define another POJO class corresponding to CERTIFICATE table so that certificate objects can be stored and retrieved into the CERTIFICATE table.

```
public class Certificate{

   private int id;

   private String name;


   public Certificate() {}

   public Certificate(String name) {

      this.name = name;

   }

   public int getId() {

      return id;

   }

   public void setId( int id ) {

      this.id = id;

   }

   public String getName() {

      return name;

   }

   public void setName( String name ) {

      this.name = name;

   }

}
```

Define Hibernate Mapping File

Let us develop our mapping file, which instructs Hibernate how to map the defined classes to the database tables. The <bag> element will be used to define the rule for the Collection used.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <bag name="certificates" cascade="all">
         <key column="employee_id"/>
         <one-to-many class="Certificate"/>
      </bag>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>

   <class name="Certificate" table="CERTIFICATE">
      <meta attribute="class-description">
         This class contains the certificate records.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="name" column="certificate_name" type="string"/>
   </class>
```

```
</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. You are already familiar with most of the mapping detail, but let us see all the elements of mapping file once again:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains two <class> elements corresponding to each class.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<bag>** element is used to set the relationship between Certificate and Employee classes. We used the **cascade** attribute in the <bag> element to tell Hibernate to persist the Certificate objects at the same time as the Employee objects. The **name** attribute is set to the defined **Collection** variable in the parent class, in our case it is *certificates*.

- The **<key>** element is the column in the CERTIFICATE table that holds the foreign key to the parent object i.e. table EMPLOYEE.

- The **<one-to-many>** element indicates that one Employee object relates to many Certificate objects and, as such, the Certificate object must have an Employee parent associated with it. You can use either **<one-to-one>**, **<many-to-one>** or **<many-to-many>** elements based on your requirement.

Create Application Class

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employees' records along with their certificates and then we will apply CRUD operations on those records.

```
import java.util.*;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {
        try{
            factory = new Configuration().configure().buildSessionFactory();
        }catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
        ManageEmployee ME = new ManageEmployee();
        /* Let us have a set of certificates for the first employee  */
        ArrayList set1 = new ArrayList();
        set1.add(new Certificate("MCA"));
        set1.add(new Certificate("MBA"));
        set1.add(new Certificate("PMP"));


        /* Add employee records in the database */
        Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, set1);


        /* Another set of certificates for the second employee  */
        ArrayList set2 = new ArrayList();
        set2.add(new Certificate("BCA"));
        set2.add(new Certificate("BA"));


        /* Add another employee record in the database */
        Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, set2);
```

```
    /* List down all the employees */
    ME.listEmployees();


    /* Update employee's salary records */
    ME.updateEmployee(empID1, 5000);


    /* Delete an employee from the database */
    ME.deleteEmployee(empID2);


    /* List down all the employees */
    ME.listEmployees();


}


/* Method to add an employee record in the database */
public Integer addEmployee(String fname, String lname,
                            int salary, ArrayList cert){
    Session session = factory.openSession();
    Transaction tx = null;
    Integer employeeID = null;
    try{
        tx = session.beginTransaction();
        Employee employee = new Employee(fname, lname, salary);
        employee.setCertificates(cert);
        employeeID = (Integer) session.save(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
    return employeeID;
}
```

```
/* Method to list all the employees detail */
public void listEmployees( ){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        List employees = session.createQuery("FROM Employee").list();
        for (Iterator iterator1 =
                        employees.iterator(); iterator1.hasNext();){
            Employee employee = (Employee) iterator1.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
            System.out.println("  Salary: " + employee.getSalary());
            Collection certificates = employee.getCertificates();
            for (Iterator iterator2 =
                        certificates.iterator(); iterator2.hasNext();){
                Certificate certName = (Certificate) iterator2.next();
                System.out.println("Certificate: " + certName.getName());
            }
        }
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}
/* Method to update salary for an employee */
public void updateEmployee(Integer EmployeeID, int salary ){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        Employee employee =
```

```
                (Employee)session.get(Employee.class, EmployeeID);
         employee.setSalary( salary );
         session.update(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to delete an employee from the records */
   public void deleteEmployee(Integer EmployeeID){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                  (Employee)session.get(Employee.class, EmployeeID);
         session.delete(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure, you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create Certificate.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result on the screen, and same time records would be created in EMPLOYEE and CERTIFICATE tables. You can see, certificates has been sorted in reverse order. You can try by changing your mapping file, simply set **sort="natural"** and execute your program and compare the results.

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Manoj  Last Name: Kumar  Salary: 4000

Certificate: MCA

Certificate: MBA

Certificate: PMP

First Name: Dilip  Last Name: Kumar  Salary: 3000

Certificate: BCA

Certificate: BA

First Name: Manoj  Last Name: Kumar  Salary: 5000

Certificate: MCA

Certificate: MBA

Certificate: PMP
```

If you check your EMPLOYEE and CERTIFICATE tables, they should have the following records:

```
mysql> select * from EMPLOYEE;

+----+------------+-----------+--------+

| id | first_name | last_name | salary |

+----+------------+-----------+--------+

| 53 | Manoj      | Kumar     |   5000 |

+----+------------+-----------+--------+

1 row in set (0.00 sec)


mysql> select * from CERTIFICATE;

+----+------------------+-------------+

| id | certificate_name | employee_id |
```

tutorialspoint
SIMPLYEASYLEARNING

```
+----+-----------------+------------+
| 11 | MCA             |         53 |
| 12 | MBA             |         53 |
| 13 | PMP             |         53 |
+----+-----------------+------------+
3 rows in set (0.00 sec)


mysql>
```

Hibernate – Map Mappings

A **Map** is a java collection that stores elements in key-value pairs and does not allow duplicate elements in the list. The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.

A Map is mapped with a <map> element in the mapping table and an unordered map can be initialized with java.util.HashMap.

Define RDBMS Tables

Consider a situation where we need to store our employee records in the EMPLOYEE table, which will have the following structure:

```
create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name  VARCHAR(20) default NULL,
    salary     INT  default NULL,
    PRIMARY KEY (id)
);
```

Further, assume each employee can have one or more certificate associated with him/her. We will store certificate related information in a separate table having the following structure:

```
create table CERTIFICATE (
    id INT NOT NULL auto_increment,
    certificate_type VARCHAR(40) default NULL,
    certificate_name VARCHAR(30) default NULL,
    employee_id INT default NULL,
    PRIMARY KEY (id)
```

```
);
```

There will be **one-to-many** relationship between EMPLOYEE and CERTIFICATE objects.

Define POJO Classes

Let us implement a POJO class **Employee,** which will be used to persist the objects related to EMPLOYEE table and having a collection of certificates in **List** variable.

```java
import java.util.*;

public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;
    private Map certificates;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
        this.lastName = lname;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
```

```
    }
    public void setLastName( String last_name ) {

       this.lastName = last_name;

    }
    public int getSalary() {

       return salary;

    }
    public void setSalary( int salary ) {

       this.salary = salary;

    }


    public Map getCertificates() {

       return certificates;

    }
    public void setCertificates( Map certificates ) {

       this.certificates = certificates;

    }
 }
```

We need to define another POJO class corresponding to CERTIFICATE table so that certificate objects can be stored and retrieved into the CERTIFICATE table.

```
 public class Certificate{
    private int id;
    private String name;

    public Certificate() {}
    public Certificate(String name) {

       this.name = name;

    }
    public int getId() {

       return id;

    }
    public void setId( int id ) {

       this.id = id;

    }
    public String getName() {
```

```
      return name;
   }
   public void setName( String name ) {
      this.name = name;
   }
}
```

Define Hibernate Mapping File

Let us develop our mapping file, which instructs Hibernate how to map the defined classes to the database tables. The <map> element will be used to define the rule for the Map used.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <map name="certificates" cascade="all">
         <key column="employee_id"/>
         <index column="certificate_type" type="string"/>
         <one-to-many class="Certificate"/>
      </map>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>


   <class name="Certificate" table="CERTIFICATE">
```

```
        <meta attribute="class-description">
           This class contains the certificate records.
        </meta>
        <id name="id" type="int" column="id">
           <generator class="native"/>
        </id>
        <property name="name" column="certificate_name" type="string"/>
    </class>


 </hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. You are already familiar with most of the mapping detail, but let us see all the elements of mapping file once again:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains two <class> elements corresponding to each class.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<map>** element is used to set the relationship between Certificate and Employee classes. We used the **cascade** attribute in the <map> element to tell Hibernate to persist the Certificate objects at the same time as the Employee objects. The **name** attribute is set to the defined **Map** variable in the parent class, in our case it is *certificates*.

- The **<index>** element is used to represents the key parts of the key/value map pair. The key will be stored in the column certificate_type using a type of string.

- The **<key>** element is the column in the CERTIFICATE table that holds the foreign key to the parent object i.e. table EMPLOYEE.

- The **<one-to-many>** element indicates that one Employee object relates to many Certificate objects and, as such, the Certificate object must have an Employee parent associated with it. You can use either **<one-to-one>**, **<many-to-one>** or **<many-to-many>** elements based on your requirement.

Create Application Class

Finally, we will create our application class with the main() method to run the application. We will use this application to save an Employee record along with a list of certificates and then we will apply CRUD operations on that record.

```java
import java.util.*;

import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new Configuration().configure().buildSessionFactory();
      }catch (Throwable ex) {
         System.err.println("Failed to create sessionFactory object." + ex);
         throw new ExceptionInInitializerError(ex);
      }
      ManageEmployee ME = new ManageEmployee();
      /* Let us have a set of certificates for the first employee  */
      HashMap set = new HashMap();
      set.put("ComputerScience", new Certificate("MCA"));
      set.put("BusinessManagement", new Certificate("MBA"));
      set.put("ProjectManagement", new Certificate("PMP"));
```

```
   /* Add employee records in the database */
   Integer empID = ME.addEmployee("Manoj", "Kumar", 4000, set);


   /* List down all the employees */
   ME.listEmployees();


   /* Update employee's salary records */
   ME.updateEmployee(empID, 5000);


   /* List down all the employees */
   ME.listEmployees();


}


/* Method to add an employee record in the database */
public Integer addEmployee(String fname, String lname,
                                int salary, HashMap cert){
   Session session = factory.openSession();
   Transaction tx = null;
   Integer employeeID = null;
   try{
      tx = session.beginTransaction();
      Employee employee = new Employee(fname, lname, salary);
      employee.setCertificates(cert);
      employeeID = (Integer) session.save(employee);
      tx.commit();
   }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   }finally {
      session.close();
   }
   return employeeID;
}
```

```
/* Method to list all the employees detail */
public void listEmployees( ){
   Session session = factory.openSession();
   Transaction tx = null;
   try{
      tx = session.beginTransaction();
      List employees = session.createQuery("FROM Employee").list();
      for (Iterator iterator1 =
                     employees.iterator(); iterator1.hasNext();){
         Employee employee = (Employee) iterator1.next();
         System.out.print("First Name: " + employee.getFirstName());
         System.out.print("  Last Name: " + employee.getLastName());
         System.out.println("  Salary: " + employee.getSalary());
         Map ec = employee.getCertificates();
         System.out.println("Certificate: " +
           (((Certificate)ec.get("ComputerScience")).getName()));
         System.out.println("Certificate: " +
           (((Certificate)ec.get("BusinessManagement")).getName()));
         System.out.println("Certificate: " +
           (((Certificate)ec.get("ProjectManagement")).getName()));
      }
      tx.commit();
   }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   }finally {
      session.close();
   }
}
/* Method to update salary for an employee */
public void updateEmployee(Integer EmployeeID, int salary ){
   Session session = factory.openSession();
   Transaction tx = null;
   try{
      tx = session.beginTransaction();
```

```
        Employee employee =
                    (Employee)session.get(Employee.class, EmployeeID);
        employee.setSalary( salary );
        session.update(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}

/* Method to delete an employee from the records */
public void deleteEmployee(Integer EmployeeID){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        Employee employee =
                    (Employee)session.get(Employee.class, EmployeeID);
        session.delete(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}
}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure, you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create Certificate.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result on the screen, and same time records would be created in EMPLOYEE and CERTIFICATE tables.

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Manoj  Last Name: Kumar  Salary: 4000

Certificate: MCA

Certificate: MBA

Certificate: PMP

First Name: Manoj  Last Name: Kumar  Salary: 5000

Certificate: MCA

Certificate: MBA

Certificate: PMP
```

If you check your EMPLOYEE and CERTIFICATE tables, they should have the following records:

```
mysql> select * from EMPLOYEE;

+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
| 60 | Manoj      | Kumar     |   5000 |
+----+------------+-----------+--------+
1 row in set (0.00 sec)


mysql>select * from CERTIFICATE;

+----+-------------------+-----------------+-------------+
| id | certificate_type  | certificate_name | employee_id |
+----+-------------------+-----------------+-------------+
| 16 | ProjectManagement | PMP              |          60 |
| 17 | BusinessManagement | MBA             |          60 |
```

tutorialspoint
SIMPLYEASYLEARNING

```
| 18 | ComputerScience   | MCA            |            60 |

+----+-------------------+----------------+-------------+

3 rows in set (0.00 sec)

mysql>
```

Hibernate – SortedMap Mappings

A **SortedMap** is a similar java collection as **Map** that stores elements in key-value pairs and provides a total ordering on its keys. Duplicate elements are not allowed in the map. The map is ordered according to the natural ordering of its keys, or by a Comparator typically provided at sorted map creation time.

A SortedMap is mapped with a <map> element in the mapping table and an ordered map can be initialized with java.util.TreeMap.

Define RDBMS Tables

Consider a situation where we need to store our employee records in EMPLOYEE table, which will have the following structure:

```
create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name  VARCHAR(20) default NULL,
    salary     INT  default NULL,
    PRIMARY KEY (id)
);
```

Further, assume each employee can have one or more certificate associated with him/her. We will store certificate related information in a separate table having the following structure:

```
create table CERTIFICATE (
    id INT NOT NULL auto_increment,
    certificate_type VARCHAR(40) default NULL,
    certificate_name VARCHAR(30) default NULL,
    employee_id INT default NULL,
    PRIMARY KEY (id)
);
```

There will be **one-to-many** relationship between EMPLOYEE and CERTIFICATE objects.

Define POJO Classes

Let us implement a POJO class Employee, which will be used to persist the objects related to EMPLOYEE table and having a collection of certificates in List variable.

```
import java.util.*;

public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;
   private SortedMap certificates;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
```

```
   }
   public int getSalary() {

      return salary;

   }
   public void setSalary( int salary ) {

      this.salary = salary;

   }


   public SortedMap getCertificates() {

      return certificates;

   }
   public void setCertificates( SortedMap certificates ) {

      this.certificates = certificates;

   }
 }
```

We need to define another POJO class corresponding to CERTIFICATE table so that certificate objects can be stored and retrieved into the CERTIFICATE table. This class should also implement Comparable interface and compareTo method, which will be used to sort the key elements of the SortedMap in case you set sort="natural" in your mapping file (see below mapping file).

```
public class Certificate implements Comparable <String>{
   private int id;
   private String name;

   public Certificate() {}
   public Certificate(String name) {

      this.name = name;

   }
   public int getId() {

      return id;

   }
   public void setId( int id ) {

      this.id = id;

   }
   public String getName() {

      return name;
```

```
    }

    public void setName( String name ) {

        this.name = name;

    }

    public int compareTo(String that){

        final int BEFORE = -1;

        final int AFTER = 1;


        if (that == null) {

            return BEFORE;

        }


        Comparable thisCertificate = this;

        Comparable thatCertificate = that;


        if(thisCertificate == null) {

            return AFTER;

        } else if(thatCertificate == null) {

            return BEFORE;

        } else {

            return thisCertificate.compareTo(thatCertificate);

        }

    }

}
```

Define Hibernate Mapping File

Let us develop our mapping file, which instructs Hibernate how to map the defined classes to the database tables. The <map> element will be used to define the rule for the Map used.

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-mapping PUBLIC

 "-//Hibernate/Hibernate Mapping DTD//EN"

 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>
```

```
    <class name="Employee" table="EMPLOYEE">

      <meta attribute="class-description">

         This class contains the employee detail.

      </meta>

      <id name="id" type="int" column="id">

         <generator class="native"/>

      </id>

      <map name="certificates" cascade="all" sort="MyClass">

         <key column="employee_id"/>

         <index column="certificate_type" type="string"/>

         <one-to-many class="Certificate"/>

      </map>

      <property name="firstName" column="first_name" type="string"/>

      <property name="lastName" column="last_name" type="string"/>

      <property name="salary" column="salary" type="int"/>

   </class>


   <class name="Certificate" table="CERTIFICATE">

      <meta attribute="class-description">

         This class contains the certificate records.

      </meta>

      <id name="id" type="int" column="id">

         <generator class="native"/>

      </id>

      <property name="name" column="certificate_name" type="string"/>

   </class>


</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. You are already familiar with most of the mapping detail, but let us see all the elements of mapping file once again:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains two <class> elements corresponding to each class.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

92

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<map>** element is used to set the relationship between Certificate and Employee classes. We used the **cascade** attribute in the <map> element to tell Hibernate to persist the Certificate objects at the same time as the Employee objects. The **name** attribute is set to the defined **SortedMap** variable in the parent class, in our case it is *certificates*. The **sort** attribute can be set to **natural** to have natural sorting or it can be set to a custom class implementing **java.util.Comparator**. We have used a class **MyClass,** which implements java.util.Comparator to reverse the sorting order implemented in **Certificate** class.

- The **<index>** element is used to represents the key parts of the key/value map pair. The key will be stored in the column certificate_type using a type of string.

- The **<key>** element is the column in the CERTIFICATE table that holds the foreign key to the parent object i.e. table EMPLOYEE.

- The **<one-to-many>** element indicates that one Employee object relates to many Certificate objects and, as such, the Certificate object must have an Employee parent associated with it. You can use either **<one-to-one>**, **<many-to-one>** or **<many-to-many>** elements based on your requirement.

If we use **sort="natural"** setting, then we do not need to create a separate class because Certificate class already has implemented Comparable interface and hibernate will use compareTo() method defined in Certificate class to compare SortedMap keys. But we are using a custom comparator class **MyClass** in our mapping file so we would have to create this class based on our sorting algorithm. Let us do descending sorting of the keys available in the map.

```
import java.util.Comparator;


public class MyClass implements Comparator <String>{
   public int compare(String o1, String o2) {
      final int BEFORE = -1;
```

```
        final int AFTER = 1;


        /* To reverse the sorting order, multiple by -1 */

        if (o2 == null) {

            return BEFORE * -1;

        }


        Comparable thisCertificate = o1;

        Comparable thatCertificate = o2;


        if(thisCertificate == null) {

            return AFTER * 1;

        } else if(thatCertificate == null) {

            return BEFORE * -1;

        } else {

            return thisCertificate.compareTo(thatCertificate) * -1;

        }

    }

}
```

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employee's records along with their certificates and then we will apply CRUD operations on those records.

```
import java.util.*;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {
        try{
            factory = new Configuration().configure().buildSessionFactory();
```

```
      }catch (Throwable ex) {
         System.err.println("Failed to create sessionFactory object." + ex);
         throw new ExceptionInInitializerError(ex);
      }
      ManageEmployee ME = new ManageEmployee();
      /* Let us have a set of certificates for the first employee  */
      TreeMap set1 = new TreeMap();
      set1.put("ComputerScience", new Certificate("MCA"));
      set1.put("BusinessManagement", new Certificate("MBA"));
      set1.put("ProjectManagement", new Certificate("PMP"));

      /* Add employee records in the database */
      Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, set1);

      /* Another set of certificates for the second employee  */
      TreeMap set2 = new TreeMap();
      set2.put("ComputerScience", new Certificate("MCA"));
      set2.put("BusinessManagement", new Certificate("MBA"));

      /* Add another employee record in the database */
      Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, set2);

      /* List down all the employees */
      ME.listEmployees();

      /* Update employee's salary records */
      ME.updateEmployee(empID1, 5000);

      /* Delete an employee from the database */
      ME.deleteEmployee(empID2);

      /* List down all the employees */
      ME.listEmployees();


   }
```

```
/* Method to add an employee record in the database */
public Integer addEmployee(String fname, String lname,
                              int salary, TreeMap cert){
   Session session = factory.openSession();
   Transaction tx = null;
   Integer employeeID = null;
   try{
      tx = session.beginTransaction();
      Employee employee = new Employee(fname, lname, salary);
      employee.setCertificates(cert);
      employeeID = (Integer) session.save(employee);
      tx.commit();
   }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   }finally {
      session.close();
   }
   return employeeID;
}


/* Method to list all the employees detail */
public void listEmployees( ){
   Session session = factory.openSession();
   Transaction tx = null;
   try{
      tx = session.beginTransaction();
      List employees = session.createQuery("FROM Employee").list();
      for (Iterator iterator1 =
                     employees.iterator(); iterator1.hasNext();){
         Employee employee = (Employee) iterator1.next();
         System.out.print("First Name: " + employee.getFirstName());
         System.out.print("  Last Name: " + employee.getLastName());
         System.out.println("  Salary: " + employee.getSalary());
```

```
        SortedMap<String, Certificate> map =
                                      employee.getCertificates();
        for(Map.Entry<String,Certificate> entry : map.entrySet()){
            System.out.print("\tCertificate Type: " +  entry.getKey());
            System.out.println(",  Name: " +
                                  (entry.getValue()).getName());

        }
     }
     tx.commit();
   }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   }finally {
      session.close();
   }
}
/* Method to update salary for an employee */
public void updateEmployee(Integer EmployeeID, int salary ){
   Session session = factory.openSession();
   Transaction tx = null;
   try{
      tx = session.beginTransaction();
      Employee employee =
               (Employee)session.get(Employee.class, EmployeeID);
      employee.setSalary( salary );
      session.update(employee);
      tx.commit();
   }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   }finally {
      session.close();
   }
}
/* Method to delete an employee from the records */
```

```
    public void deleteEmployee(Integer EmployeeID){

        Session session = factory.openSession();

        Transaction tx = null;

        try{

            tx = session.beginTransaction();

            Employee employee =

                    (Employee)session.get(Employee.class, EmployeeID);

            session.delete(employee);

            tx.commit();

        }catch (HibernateException e) {

            if (tx!=null) tx.rollback();

            e.printStackTrace();

        }finally {

            session.close();

        }

    }

}
```

Compilation and Execution

Here are the steps to compile and run the above application. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create Certificate.java source file as shown above and compile it.

- Create MyClass.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result on the screen, and same time, records would be created in EMPLOYEE and CERTIFICATE tables. You can see, certificates type has been sorted in reverse order. You can try by changing your mapping file, simply set **sort="natural"** and execute your program and compare the results.

```
$java ManageEmployee
.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........
```

```
First Name: Manoj  Last Name: Kumar  Salary: 4000
     Certificate Type: ProjectManagement,  Name: PMP
     Certificate Type: ComputerScience,  Name: MCA
     Certificate Type: BusinessManagement,  Name: MBA
First Name: Dilip  Last Name: Kumar  Salary: 3000
     Certificate Type: ComputerScience,  Name: MCA
     Certificate Type: BusinessManagement,  Name: MBA
First Name: Manoj  Last Name: Kumar  Salary: 5000
     Certificate Type: ProjectManagement,  Name: PMP
     Certificate Type: ComputerScience,  Name: MCA
     Certificate Type: BusinessManagement,  Name: MBA
```

If you check your EMPLOYEE and CERTIFICATE tables, they should have the following records:

```
mysql> select * from EMPLOYEE;

+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
| 74 | Manoj      | Kumar     |   5000 |
+----+------------+-----------+--------+
1 row in set (0.00 sec)


mysql> select * from CERTIFICATE;

+----+--------------------+------------------+-------------+
| id | certificate_type   | certificate_name | employee_id |
+----+--------------------+------------------+-------------+
| 52 | BusinessManagement | MBA              |          74 |
| 53 | ComputerScience    | MCA              |          74 |
| 54 | ProjectManagement  | PMP              |          74 |
+----+--------------------+------------------+-------------+
3 rows in set (0.00 sec)


mysql>
```

Association Mappings

The mapping of associations between entity classes and the relationships between tables is the soul of ORM. Following are the four ways in which the cardinality of the relationship between the objects can be expressed. An association mapping can be unidirectional as well as bidirectional.

Mapping type	Description
Many-to-One	Mapping many-to-one relationship using Hibernate
One-to-One	Mapping one-to-one relationship using Hibernate
One-to-Many	Mapping one-to-many relationship using Hibernate
Many-to-Many	Mapping many-to-many relationship using Hibernate

Hibernate – Many-to-One Mappings

A **many-to-one** association is the most common kind of association where an Object can be associated with multiple objects. For example, the same address object can be associated with multiple employee objects.

Define RDBMS Tables

Consider a situation where we need to store our employee records in EMPLOYEE table, which will have the following structure:

```
create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name  VARCHAR(20) default NULL,
    salary     INT  default NULL,
    address    INT NOT NULL,
    PRIMARY KEY (id)
);
```

Further, many employee can have same address, so this association can be presented using many-to-one association. We will store address related information in a separate table having the following structure:

```
create table ADDRESS (
    id INT NOT NULL auto_increment,
    street_name VARCHAR(40) default NULL,
    city_name VARCHAR(40) default NULL,
    state_name VARCHAR(40) default NULL,
```

```
    zipcode VARCHAR(10) default NULL,
    PRIMARY KEY (id)
);
```

Create both the RBDMS tables and keep them ready for the next implementation.

Define POJO Classes

Let us implement a POJO class **Employee,** which will be used to persist the objects related to EMPLOYEE table and having a variable of Address type.

```java
import java.util.*;

public class Employee{
   private int id;
   private String firstName;
   private String lastName;
   private int salary;
   private Address address;

   public Employee() {}
   public Employee(String fname, String lname,
                   int salary, Address address ) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
      this.address = address;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
```

```
      this.firstName = first_name;

   }

   public String getLastName() {

      return lastName;

   }

   public void setLastName( String last_name ) {

      this.lastName = last_name;

   }

   public int getSalary() {

      return salary;

   }

   public void setSalary( int salary ) {

      this.salary = salary;

   }


   public Address getAddress() {

      return address;

   }

   public void setAddress( Address address ) {

      this.address = address;

   }

}
```

We need to define another POJO class corresponding to ADDRESS table so that address objects can be stored and retrieved into the ADDRESS table.

```
import java.util.*;


public class Address{

   private int id;

   private String street;

   private String city;

   private String state;

   private String zipcode;


   public Address() {}

   public Address(String street, String city,
```

```
               String state, String zipcode) {
   this.street = street;
   this.city = city;
   this.state = state;
   this.zipcode = zipcode;
}
public int getId() {
   return id;
}
public void setId( int id ) {
   this.id = id;
}
public String getStreet() {
   return street;
}
public void setStreet( String street ) {
   this.street = street;
}
public String getCity() {
   return city;
}
public void setCity( String city ) {
   this.city = city;
}
public String getState() {
   return state;
}
public void setState( String state ) {
   this.state = state;
}
public String getZipcode() {
   return zipcode;
}
public void setZipcode( String zipcode ) {
   this.zipcode = zipcode;
}
```

```
    }
```

Define Hibernate Mapping File

Let us develop our mapping files, which instructs Hibernate how to map the defined classes to the database tables. The <many-to-one> element will be used to define the rule to establish a many-to-one relationship between EMPLOYEE and ADDRESS entities.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
      <many-to-one name="address" column="address"
       class="Address" not-null="true"/>
   </class>

   <class name="Address" table="ADDRESS">
      <meta attribute="class-description">
         This class contains the address detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="street" column="street_name" type="string"/>
      <property name="city" column="city_name" type="string"/>
```

```
        <property name="state" column="state_name" type="string"/>

        <property name="zipcode" column="zipcode" type="string"/>

    </class>


</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. You are already familiar with most of the mapping detail, but let us see all the elements of mapping file once again:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains two <class> elements corresponding to each class.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<many-to-one>** element is used to set the relationship between EMPLOYEE and ADDRESS entities. The **name** attribute is set to the defined variable in the parent class, in our case it is *address*. The **column** attribute is used to set the column name in the parent table EMPLOYEE.

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employee's records along with their addresses and then we will apply CRUD operations on those records.

tutorialspoint
SIMPLYEASYLEARNING

```
import java.util.*;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {

    private static SessionFactory factory;

    public static void main(String[] args) {

        try{

            factory = new Configuration().configure().buildSessionFactory();

        }catch (Throwable ex) {

            System.err.println("Failed to create sessionFactory object." + ex);

            throw new ExceptionInInitializerError(ex);

        }

        ManageEmployee ME = new ManageEmployee();


        /* Let us have one address object */

        Address address = ME.addAddress("Kondapur","Hyderabad","AP","532");


        /* Add employee records in the database */

        Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, address);


        /* Add another employee record in the database */

        Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, address);


        /* List down all the employees */

        ME.listEmployees();


        /* Update employee's salary records */

        ME.updateEmployee(empID1, 5000);


        /* Delete an employee from the database */
```

```
      ME.deleteEmployee(empID2);


      /* List down all the employees */

      ME.listEmployees();


   }


   /* Method to add an address record in the database */

   public Address addAddress(String street, String city,

                            String state, String zipcode) {

      Session session = factory.openSession();

      Transaction tx = null;

      Integer addressID = null;

      Address address = null;

      try{

         tx = session.beginTransaction();

         address = new Address(street, city, state, zipcode);

         addressID = (Integer) session.save(address);

         tx.commit();

      }catch (HibernateException e) {

         if (tx!=null) tx.rollback();

         e.printStackTrace();

      }finally {

         session.close();

      }

      return address;

   }


   /* Method to add an employee record in the database */

   public Integer addEmployee(String fname, String lname,

                              int salary, Address address){

      Session session = factory.openSession();

      Transaction tx = null;

      Integer employeeID = null;

      try{
```

```
        tx = session.beginTransaction();

        Employee employee = new Employee(fname, lname, salary, address);

        employeeID = (Integer) session.save(employee);

        tx.commit();

    }catch (HibernateException e) {

        if (tx!=null) tx.rollback();

        e.printStackTrace();

    }finally {

        session.close();

    }

    return employeeID;

}


/* Method to list all the employees detail */

public void listEmployees( ){

    Session session = factory.openSession();

    Transaction tx = null;

    try{

        tx = session.beginTransaction();

        List employees = session.createQuery("FROM Employee").list();

        for (Iterator iterator =

                         employees.iterator(); iterator.hasNext();){

            Employee employee = (Employee) iterator.next();

            System.out.print("First Name: " + employee.getFirstName());

            System.out.print("  Last Name: " + employee.getLastName());

            System.out.println("  Salary: " + employee.getSalary());

            Address add = employee.getAddress();

            System.out.println("Address ");

            System.out.println("\tStreet: " +  add.getStreet());

            System.out.println("\tCity: " + add.getCity());

            System.out.println("\tState: " + add.getState());

            System.out.println("\tZipcode: " + add.getZipcode());

        }

        tx.commit();

    }catch (HibernateException e) {
```

```
      if (tx!=null) tx.rollback();

      e.printStackTrace();

   }finally {

      session.close();

   }

}
/* Method to update salary for an employee */

public void updateEmployee(Integer EmployeeID, int salary ){

   Session session = factory.openSession();

   Transaction tx = null;

   try{

      tx = session.beginTransaction();

      Employee employee =
                (Employee)session.get(Employee.class, EmployeeID);

      employee.setSalary( salary );

      session.update(employee);

      tx.commit();

   }catch (HibernateException e) {

      if (tx!=null) tx.rollback();

      e.printStackTrace();

   }finally {

      session.close();

   }

}
/* Method to delete an employee from the records */

public void deleteEmployee(Integer EmployeeID){

   Session session = factory.openSession();

   Transaction tx = null;

   try{

      tx = session.beginTransaction();

      Employee employee =
                (Employee)session.get(Employee.class, EmployeeID);

      session.delete(employee);

      tx.commit();

   }catch (HibernateException e) {
```

```
        if (tx!=null) tx.rollback();

        e.printStackTrace();

    }finally {

        session.close();

    }

  }

}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create Address.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result on the screen, and same time records would be created in EMPLOYEE and ADDRESS tables.

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Manoj  Last Name: Kumar  Salary: 4000

Address

      Street: Kondapur

      City: Hyderabad

      State: AP

      Zipcode: 532

First Name: Dilip  Last Name: Kumar  Salary: 3000

Address

      Street: Kondapur

      City: Hyderabad

      State: AP

      Zipcode: 532
```

```
First Name: Manoj   Last Name: Kumar   Salary: 5000
Address
        Street: Kondapur
        City: Hyderabad
        State: AP
        Zipcode: 532
```

If you check your EMPLOYEE and ADDRESS tables, they should have the following records:

```
mysql> select * from EMPLOYEE;

+----+------------+-----------+--------+---------+
| id | first_name | last_name | salary | address |
+----+------------+-----------+--------+---------+
|  1 | Manoj      | Kumar     |   5000 | 5       |
+----+------------+-----------+--------+---------+
1 row in set (0.00 sec)


mysql> select * from ADDRESS;

+----+------------+-----------+------------+---------+
| id | street_name | city_name | state_name | zipcode |
+----+------------+-----------+------------+---------+
|  1 | Kondapur    | Hyderabad | AP         | 532     |
+----+------------+-----------+------------+---------+
1 row in set (0.00 sec)


mysql>
```

Hibernate – One-to-One Mappings

A **one-to-one** association is similar to **many-to-one** association with a difference that the column will be set as unique. For example, an address object can be associated with a single employee object.

Define RDBMS Tables

Consider a situation where we need to store our employee records in EMPLOYEE table, which will have the following structure:

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
```

```
    first_name VARCHAR(20) default NULL,

    last_name  VARCHAR(20) default NULL,

    salary     INT  default NULL,

    address    INT NOT NULL,

    PRIMARY KEY (id)

);
```

Further, assuming that an address can be associated a single employee only, so this association can be presented using one-to-one association. We will store address related information in a separate table having the following structure:

```
create table ADDRESS (

    id INT NOT NULL auto_increment,

    street_name VARCHAR(40) default NULL,

    city_name VARCHAR(40) default NULL,

    state_name VARCHAR(40) default NULL,

    zipcode VARCHAR(10) default NULL,

    PRIMARY KEY (id)

);
```

Create both the RBDMS tables and keep them ready for the next implementation.

Define POJO Classes

Let us implement a POJO class **Employee,** which will be used to persist the objects related to EMPLOYEE table and having a variable of Address type.

```
import java.util.*;


public class Employee{

    private int id;

    private String firstName;

    private String lastName;

    private int salary;

    private Address address;


    public Employee() {}

    public Employee(String fname, String lname,

                    int salary, Address address ) {

        this.firstName = fname;
```

```
      this.lastName = lname;

      this.salary = salary;

      this.address = address;

   }

   public int getId() {

      return id;

   }

   public void setId( int id ) {

      this.id = id;

   }

   public String getFirstName() {

      return firstName;

   }

   public void setFirstName( String first_name ) {

      this.firstName = first_name;

   }

   public String getLastName() {

      return lastName;

   }

   public void setLastName( String last_name ) {

      this.lastName = last_name;

   }

   public int getSalary() {

      return salary;

   }

   public void setSalary( int salary ) {

      this.salary = salary;

   }


   public Address getAddress() {

      return address;

   }

   public void setAddress( Address address ) {

      this.address = address;

   }
```

```
}
```

We need to define another POJO class corresponding to ADDRESS table so that address objects can be stored and retrieved into the ADDRESS table.

```java
import java.util.*;

public class Address{
   private int id;
   private String street;
   private String city;
   private String state;
   private String zipcode;

   public Address() {}
   public Address(String street, String city,
               String state, String zipcode) {
      this.street = street;
      this.city = city;
      this.state = state;
      this.zipcode = zipcode;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getStreet() {
      return street;
   }
   public void setStreet( String street ) {
      this.street = street;
   }
   public String getCity() {
      return city;
   }
}
```

```
      public void setCity( String city ) {
         this.city = city;
      }

      public String getState() {
         return state;
      }

      public void setState( String state ) {
         this.state = state;
      }

      public String getZipcode() {
         return zipcode;
      }

      public void setZipcode( String zipcode ) {
         this.zipcode = zipcode;
      }

}
```

Define Hibernate Mapping File

Let us develop our mapping files which instructs Hibernate how to map the defined classes to the database tables. The <many-to-one> element will be used to define the rule to establish a one-to-one relationship between EMPLOYEE and ADDRESS entities, but column attribute will be set to **unique** constraint and rest of the mapping file will remain as it was in case of many-to-one association.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
```

```
        </id>

        <property name="firstName" column="first_name" type="string"/>

        <property name="lastName" column="last_name" type="string"/>

        <property name="salary" column="salary" type="int"/>

        <many-to-one name="address" column="address" unique="true"

         class="Address" not-null="true"/>

    </class>


    <class name="Address" table="ADDRESS">

        <meta attribute="class-description">

            This class contains the address detail.

        </meta>

        <id name="id" type="int" column="id">

            <generator class="native"/>

        </id>

        <property name="street" column="street_name" type="string"/>

        <property name="city" column="city_name" type="string"/>

        <property name="state" column="state_name" type="string"/>

        <property name="zipcode" column="zipcode" type="string"/>

    </class>


</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. You are already familiar with most of the mapping detail, but let us see all the elements of mapping file once again:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element which contains two <class> elements corresponding to each class.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<many-to-one>** element is used to set the relationship between EMPLOYEE and ADDRESS entities. The **name** attribute is set to the defined variable in the parent class, in our case it is *address*. The **column** attribute is used to set the column name in the parent table EMPLOYEE, which is set to **unique** so that only one Employee object can be associated with an address object.

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employee's records along with their certificates and then we will apply CRUD operations on those records.

```
import java.util.*;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new Configuration().configure().buildSessionFactory();
      }catch (Throwable ex) {
         System.err.println("Failed to create sessionFactory object." + ex);
         throw new ExceptionInInitializerError(ex);
      }
      ManageEmployee ME = new ManageEmployee();


      /* Let us have one address object */
      Address address1 = ME.addAddress("Kondapur","Hyderabad","AP","532");
```

```
   /* Add employee records in the database */
   Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, address1);


   /* Let us have another address object */
   Address address2 = ME.addAddress("Saharanpur","Ambehta","UP","111");


 /* Add another employee record in the database */
   Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, address2);


   /* List down all the employees */
   ME.listEmployees();


   /* Update employee's salary records */
   ME.updateEmployee(empID1, 5000);


   /* List down all the employees */
   ME.listEmployees();


}


/* Method to add an address record in the database */
public Address addAddress(String street, String city,
                          String state, String zipcode) {
   Session session = factory.openSession();
   Transaction tx = null;
   Integer addressID = null;
   Address address = null;
   try{
      tx = session.beginTransaction();
      address = new Address(street, city, state, zipcode);
      addressID = (Integer) session.save(address);
      tx.commit();
   }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
```

```
   }finally {
      session.close();
   }
   return address;
}


/* Method to add an employee record in the database */
public Integer addEmployee(String fname, String lname,
                           int salary, Address address){
   Session session = factory.openSession();
   Transaction tx = null;
   Integer employeeID = null;
   try{
      tx = session.beginTransaction();
      Employee employee = new Employee(fname, lname, salary, address);
      employeeID = (Integer) session.save(employee);
      tx.commit();
   }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   }finally {
      session.close();
   }
   return employeeID;
}


/* Method to list all the employees detail */
public void listEmployees( ){
   Session session = factory.openSession();
   Transaction tx = null;
   try{
      tx = session.beginTransaction();
      List employees = session.createQuery("FROM Employee").list();
      for (Iterator iterator =
                     employees.iterator(); iterator.hasNext();){
```

```
            Employee employee = (Employee) iterator.next();

            System.out.print("First Name: " + employee.getFirstName());

            System.out.print("  Last Name: " + employee.getLastName());

            System.out.println("  Salary: " + employee.getSalary());

            Address add = employee.getAddress();

            System.out.println("Address ");

            System.out.println("\tStreet: " +  add.getStreet());

            System.out.println("\tCity: " + add.getCity());

            System.out.println("\tState: " + add.getState());

            System.out.println("\tZipcode: " + add.getZipcode());

         }
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to update salary for an employee */
   public void updateEmployee(Integer EmployeeID, int salary ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                    (Employee)session.get(Employee.class, EmployeeID);
         employee.setSalary( salary );
         session.update(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
```

```
        }
    }
}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create Address.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result on the screen, and at the same time, records would be created in EMPLOYEE and ADDRESS tables.

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Manoj  Last Name: Kumar  Salary: 4000
Address
        Street: Kondapur
        City: Hyderabad
        State: AP
        Zipcode: 532
First Name: Dilip  Last Name: Kumar  Salary: 3000
Address
        Street: Saharanpur
        City: Ambehta
        State: UP
        Zipcode: 111
First Name: Manoj  Last Name: Kumar  Salary: 5000
Address
        Street: Kondapur
        City: Hyderabad
```

```
       State: AP

       Zipcode: 532

First Name: Dilip  Last Name: Kumar  Salary: 3000

Address

       Street: Saharanpur

       City: Ambehta

       State: UP

       Zipcode: 111
```

If you check your EMPLOYEE and CERTIFICATE tables, they should have the following records:

```
mysql> select * from EMPLOYEE;

+----+------------+-----------+--------+---------+
| id | first_name | last_name | salary | address |
+----+------------+-----------+--------+---------+
|  7 | Manoj      | Kumar     |   5000 |       5 |
|  8 | Dilip      | Kumar     |   3000 |       6 |
+----+------------+-----------+--------+---------+
2 rows in set (0.00 sec)


mysql> select * from ADDRESS;

+----+-------------+-----------+------------+---------+
| id | street_name | city_name | state_name | zipcode |
+----+-------------+-----------+------------+---------+
|  5 | Kondapur    | Hyderabad | AP         | 532     |
|  6 | Saharanpur  | Ambehta   | UP         | 111     |
+----+-------------+-----------+------------+---------+
2 rows in set (0.00 sec)


mysql>
```

Hibernate – One-to-Many Mappings

A **One-to-Many** mapping can be implemented using a **Set** java collection that does not contain any duplicate element. We already have seen how to map **Set** collection in hibernate, so if you already learned **Set** mapping then you are all set to go with one-to-many mapping.

A Set is mapped with a <set> element in the mapping table and initialized with java.util.HashSet. You can use Set collection in your class when there is no duplicate element required in the collection.

Define RDBMS Tables

Consider a situation where we need to store our employee records in EMPLOYEE table, which will have the following structure:

```
create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name  VARCHAR(20) default NULL,
    salary     INT  default NULL,
    PRIMARY KEY (id)
);
```

Further, assume each employee can have one or more certificate associated with him/her. So, we will store certificate related information in a separate table having the following structure:

```
create table CERTIFICATE (
    id INT NOT NULL auto_increment,
    certificate_name VARCHAR(30) default NULL,
    employee_id INT default NULL,
    PRIMARY KEY (id)
);
```

There will be **one-to-many** relationship between EMPLOYEE and CERTIFICATE objects:

Define POJO Classes

Let us implement our POJO class **Employee,** which will be used to persist the objects related to EMPLOYEE table and having a collection of certificates in a **Set** variable.

```
import java.util.*;

public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;
    private Set certificates;
```

```
public Employee() {}
public Employee(String fname, String lname, int salary) {
    this.firstName = fname;
    this.lastName = lname;
    this.salary = salary;
}
public int getId() {
    return id;
}
public void setId( int id ) {
    this.id = id;
}
public String getFirstName() {
    return firstName;
}
public void setFirstName( String first_name ) {
    this.firstName = first_name;
}
public String getLastName() {
    return lastName;
}
public void setLastName( String last_name ) {
    this.lastName = last_name;
}
public int getSalary() {
    return salary;
}
public void setSalary( int salary ) {
    this.salary = salary;
}

public Set getCertificates() {
    return certificates;
}
```

```
    public void setCertificates( Set certificates ) {

        this.certificates = certificates;

    }

}
```

Now, let us define another POJO class corresponding to CERTIFICATE table so that certificate objects can be stored and retrieved into the CERTIFICATE table. This class should also implement both the equals() and hashCode() methods so that Java can determine whether any two elements/objects are identical.

```
public class Certificate {

    private int id;

    private String name;


    public Certificate() {}

    public Certificate(String name) {

        this.name = name;

    }

    public int getId() {

        return id;

    }

    public void setId( int id ) {

        this.id = id;

    }

    public String getName() {

        return name;

    }

    public void setName( String name ) {

        this.name = name;

    }

    public boolean equals(Object obj) {

        if (obj == null) return false;

        if (!this.getClass().equals(obj.getClass())) return false;


        Certificate obj2 = (Certificate)obj;

        if((this.id == obj2.getId()) && (this.name.equals(obj2.getName())))

        {
```

125

```
            return true;
        }
        return false;
    }
    public int hashCode() {
        int tmp = 0;
        tmp = ( id + name ).hashCode();
        return tmp;
    }
}
```

Define Hibernate Mapping File

Let us develop our mapping file, which instructs Hibernate how to map the defined classes to the database tables.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <set name="certificates" cascade="all">
         <key column="employee_id"/>
         <one-to-many class="Certificate"/>
      </set>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>
```

```
    <class name="Certificate" table="CERTIFICATE">

      <meta attribute="class-description">

         This class contains the certificate records.

      </meta>

      <id name="id" type="int" column="id">

         <generator class="native"/>

      </id>

      <property name="name" column="certificate_name" type="string"/>

    </class>


</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. You are already familiar with most of the mapping detail, but let us see all the elements of mapping file once again:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains two <class> elements corresponding to each class.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<set>** element sets the relationship between Certificate and Employee classes. We used the **cascade** attribute in the <set> element to tell Hibernate to persist the Certificate objects at the same time as the Employee objects. The **name**

127

attribute is set to the defined **Set** variable in the parent class, in our case it is *certificates*. For each set variable, we need to define a separate set element in the mapping file.

- <mark>The **<key>** element is the column in the CERTIFICATE table that holds the foreign key to the parent object i.e. table EMPLOYEE.</mark>

- <mark>The **<one-to-many>** element indicates that one Employee object relates to many Certificate objects.</mark>

Create Application Class

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employee's records along with their certificates and then we will apply CRUD operations on those records.

```java
import java.util.*;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new Configuration().configure().buildSessionFactory();
      }catch (Throwable ex) {
         System.err.println("Failed to create sessionFactory object." + ex);
         throw new ExceptionInInitializerError(ex);
      }
      ManageEmployee ME = new ManageEmployee();
      /* Let us have a set of certificates for the first employee  */
      HashSet set1 = new HashSet();
      set1.add(new Certificate("MCA"));
      set1.add(new Certificate("MBA"));
      set1.add(new Certificate("PMP"));


      /* Add employee records in the database */
```

```
    Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, set1);


    /* Another set of certificates for the second employee  */
    HashSet set2 = new HashSet();
    set2.add(new Certificate("BCA"));
    set2.add(new Certificate("BA"));


    /* Add another employee record in the database */
    Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, set2);


    /* List down all the employees */
    ME.listEmployees();


    /* Update employee's salary records */
    ME.updateEmployee(empID1, 5000);


    /* Delete an employee from the database */
    ME.deleteEmployee(empID2);


    /* List down all the employees */
    ME.listEmployees();

}


/* Method to add an employee record in the database */
public Integer addEmployee(String fname, String lname,
                                    int salary, Set cert){
    Session session = factory.openSession();
    Transaction tx = null;
    Integer employeeID = null;
    try{
        tx = session.beginTransaction();
        Employee employee = new Employee(fname, lname, salary);
        employee.setCertificates(cert);
        employeeID = (Integer) session.save(employee);
```

```
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
    return employeeID;
}


/* Method to list all the employees detail */
public void listEmployees( ){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        List employees = session.createQuery("FROM Employee").list();
        for (Iterator iterator1 =
                        employees.iterator(); iterator1.hasNext();){
            Employee employee = (Employee) iterator1.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
            System.out.println("  Salary: " + employee.getSalary());
            Set certificates = employee.getCertificates();
            for (Iterator iterator2 =
                        certificates.iterator(); iterator2.hasNext();){
                Certificate certName = (Certificate) iterator2.next();
                System.out.println("Certificate: " + certName.getName());
            }
        }
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
```

```
          session.close();
       }
   }
   /* Method to update salary for an employee */
   public void updateEmployee(Integer EmployeeID, int salary ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                    (Employee)session.get(Employee.class, EmployeeID);
         employee.setSalary( salary );
         session.update(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to delete an employee from the records */
   public void deleteEmployee(Integer EmployeeID){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                    (Employee)session.get(Employee.class, EmployeeID);
         session.delete(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
```

```
        session.close();
      }
    }
}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create Certificate.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result on the screen, and same time records would be created in EMPLOYEE and CERTIFICATE tables.

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Manoj  Last Name: Kumar  Salary: 4000

Certificate: MBA

Certificate: PMP

Certificate: MCA

First Name: Dilip  Last Name: Kumar  Salary: 3000

Certificate: BCA

Certificate: BA

First Name: Manoj  Last Name: Kumar  Salary: 5000

Certificate: MBA

Certificate: PMP

Certificate: MCA
```

If you check your EMPLOYEE and CERTIFICATE tables, they should have the following records:

```
mysql> select * from employee;
```

```
+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
| 1  | Manoj      | Kumar     |   5000 |
+----+------------+-----------+--------+
1 row in set (0.00 sec)


mysql> select * from certificate;
+----+------------------+-------------+
| id | certificate_name | employee_id |
+----+------------------+-------------+
| 1  | MBA              |           1 |
| 2  | PMP              |           1 |
| 3  | MCA              |           1 |
+----+------------------+-------------+
3 rows in set (0.00 sec)


mysql>
```

Hibernate – Many-to-Many Mappings

A **Many-to-Many** mapping can be implemented using a **Set** java collection that does not contain any duplicate element. We already have seen how to map **Set** collection in hibernate, so if you already learned **Set** mapping, then you are all set to go with many-to-many mapping.

A Set is mapped with a <set> element in the mapping table and initialized with java.util.HashSet. You can use Set collection in your class when there is no duplicate element required in the collection.

Define RDBMS Tables

Consider a situation where we need to store our employee records in EMPLOYEE table, which will have the following structure:

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
```

```
    PRIMARY KEY (id)
);
```

Further, assume each employee can have one or more certificate associated with him/her and a similar certificate can be associated with more than one employee. We will store certificate related information in a separate table, which has the following structure:

```
create table CERTIFICATE (
    id INT NOT NULL auto_increment,
    certificate_name VARCHAR(30) default NULL,
    PRIMARY KEY (id)
);
```

Now to implement **many-to-many** relationship between EMPLOYEE and CERTIFICATE objects, we would have to introduce one more intermediate table having Employee ID and Certificate ID as follows:

```
create table EMP_CERT (
    employee_id INT NOT NULL,
    certificate_id INT NOT NULL,
    PRIMARY KEY (employee_id,certificate_id)
);
```

Define POJO Classes

Let us implement our POJO class **Employee,** which will be used to persist the objects related to EMPLOYEE table and having a collection of certificates in **Set** variable.

```
import java.util.*;

public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;
    private Set certificates;


    public Employee() {}
```

```
public Employee(String fname, String lname, int salary) {

   this.firstName = fname;

   this.lastName = lname;

   this.salary = salary;

}

public int getId() {

   return id;

}

public void setId( int id ) {

   this.id = id;

}

public String getFirstName() {

   return firstName;

}

public void setFirstName( String first_name ) {

   this.firstName = first_name;

}

public String getLastName() {

   return lastName;

}

public void setLastName( String last_name ) {

   this.lastName = last_name;

}

public int getSalary() {

   return salary;

}

public void setSalary( int salary ) {

   this.salary = salary;

}


public Set getCertificates() {

   return certificates;

}

public void setCertificates( Set certificates ) {

   this.certificates = certificates;
```

```
      }
 }
```

Now let us define another POJO class corresponding to CERTIFICATE table so that certificate objects can be stored and retrieved into the CERTIFICATE table. This class should also implement both the equals() and hashCode() methods so that Java can determine whether any two elements/objects are identical.

```java
public class Certificate {
    private int id;
    private String name;

    public Certificate() {}
    public Certificate(String name) {
        this.name = name;
    }
    public int getId() {
        return id;
    }
    public void setId( int id ) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName( String name ) {
        this.name = name;
    }
    public boolean equals(Object obj) {
        if (obj == null) return false;
        if (!this.getClass().equals(obj.getClass())) return false;

        Certificate obj2 = (Certificate)obj;
        if((this.id == obj2.getId()) && (this.name.equals(obj2.getName())))
        {
            return true;
        }
    }
```

```
        return false;
    }
    public int hashCode() {
        int tmp = 0;
        tmp = ( id + name ).hashCode();
        return tmp;
    }
}
```

Define Hibernate Mapping File

Let us develop our mapping file, which instructs Hibernate — how to map the defined classes to the database tables. The <set> element will be used to define the rule for many-to-many relationship.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>
    <class name="Employee" table="EMPLOYEE">
        <meta attribute="class-description">
            This class contains the employee detail.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="native"/>
        </id>
        <set name="certificates" cascade="save-update" table="EMP_CERT">
            <key column="employee_id"/>
            <many-to-many column="certificate_id" class="Certificate"/>
        </set>
        <property name="firstName" column="first_name" type="string"/>
        <property name="lastName" column="last_name" type="string"/>
        <property name="salary" column="salary" type="int"/>
    </class>
```

```
   <class name="Certificate" table="CERTIFICATE">

     <meta attribute="class-description">

        This class contains the certificate records.

     </meta>

     <id name="id" type="int" column="id">

        <generator class="native"/>

     </id>

     <property name="name" column="certificate_name" type="string"/>

   </class>


</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. You are already familiar with most of the mapping detail, but let us see all the elements of mapping file once again:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains two <class> elements corresponding to each class.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<set>** element sets the relationship between Certificate and Employee classes. We set **cascade** attribute to **save-update** to tell Hibernate to persist the Certificate objects for SAVE i.e. CREATE and UPDATE operations at the same time as the Employee objects. The **name** attribute is set to the defined **Set** variable in the parent class, in our case it is *certificates*. For each set variable, we need to

138

define a separate set element in the mapping file. Here we used **name** attribute to set the intermediate table name to EMP_CERT.

- The **<key>** element is the column in the EMP_CERT table that holds the foreign key to the parent object i.e. table EMPLOYEE and links to the certification_id in the CERTIFICATE table.

- The **<many-to-many>** element indicates that one Employee object relates to many Certificate objects and column attributes are used to link intermediate EMP_CERT.

Create Application Class

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employee's records along with their certificates and then we will apply CRUD operations on those records.

```java
import java.util.*;

import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new Configuration().configure().buildSessionFactory();
      }catch (Throwable ex) {
         System.err.println("Failed to create sessionFactory object." + ex);
         throw new ExceptionInInitializerError(ex);
      }
      ManageEmployee ME = new ManageEmployee();
      /* Let us have a set of certificates for the first employee  */
      HashSet certificates = new HashSet();

      certificates.add(new Certificate("MCA"));
      certificates.add(new Certificate("MBA"));

      certificates.add(new Certificate("PMP"));
```

```
   /* Add employee records in the database */
   Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, certificates);


   /* Add another employee record in the database */
   Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, certificates);


   /* List down all the employees */
   ME.listEmployees();


   /* Update employee's salary records */
   ME.updateEmployee(empID1, 5000);


   /* Delete an employee from the database */
   ME.deleteEmployee(empID2);


   /* List down all the employees */
   ME.listEmployees();

}


/* Method to add an employee record in the database */
public Integer addEmployee(String fname, String lname,
                                      int salary, Set cert){
   Session session = factory.openSession();
   Transaction tx = null;
   Integer employeeID = null;
   try{
      tx = session.beginTransaction();
      Employee employee = new Employee(fname, lname, salary);
      employee.setCertificates(cert);
      employeeID = (Integer) session.save(employee);
      tx.commit();
   }catch (HibernateException e) {

      if (tx!=null) tx.rollback();
```

```
            e.printStackTrace();
        }finally {
            session.close();
        }
        return employeeID;
    }


    /* Method to list all the employees detail */
    public void listEmployees( ){
        Session session = factory.openSession();
        Transaction tx = null;
        try{
            tx = session.beginTransaction();
            List employees = session.createQuery("FROM Employee").list();
            for (Iterator iterator1 =
                            employees.iterator(); iterator1.hasNext();){
                Employee employee = (Employee) iterator1.next();
                System.out.print("First Name: " + employee.getFirstName());
                System.out.print("  Last Name: " + employee.getLastName());
                System.out.println("  Salary: " + employee.getSalary());
                Set certificates = employee.getCertificates();
                for (Iterator iterator2 =
                            certificates.iterator(); iterator2.hasNext();){
                    Certificate certName = (Certificate) iterator2.next();
                    System.out.println("Certificate: " + certName.getName());
                }
            }
            tx.commit();
        }catch (HibernateException e) {
            if (tx!=null) tx.rollback();
            e.printStackTrace();
        }finally {
            session.close();
        }

    }
```

```
/* Method to update salary for an employee */
public void updateEmployee(Integer EmployeeID, int salary ){
   Session session = factory.openSession();
   Transaction tx = null;
   try{
      tx = session.beginTransaction();
      Employee employee =
                (Employee)session.get(Employee.class, EmployeeID);
      employee.setSalary( salary );
      session.update(employee);
      tx.commit();
   }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   }finally {
      session.close();
   }
}
/* Method to delete an employee from the records */
public void deleteEmployee(Integer EmployeeID){
   Session session = factory.openSession();
   Transaction tx = null;
   try{
      tx = session.beginTransaction();
      Employee employee =
                (Employee)session.get(Employee.class, EmployeeID);
      session.delete(employee);
      tx.commit();
   }catch (HibernateException e) {
      if (tx!=null) tx.rollback();
      e.printStackTrace();
   }finally {
      session.close();
   }
}
```

```
}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create Certificate.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result on the screen, and same time records would be created in EMPLOYEE, EMP_CERT and CERTIFICATE tables.

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Manoj  Last Name: Kumar  Salary: 4000

Certificate: MBA

Certificate: PMP

Certificate: MCA

First Name: Dilip  Last Name: Kumar  Salary: 3000

Certificate: MBA

Certificate: PMP

Certificate: MCA

First Name: Manoj  Last Name: Kumar  Salary: 5000

Certificate: MBA

Certificate: PMP

Certificate: MCA
```

If you check your EMPLOYEE, EMP_CERT and CERTIFICATE tables, they should have the following records:

```
mysql> select * from EMPLOYEE;

+----+------------+-----------+--------+

| id | first_name | last_name | salary |
```

```
+----+-----------+-----------+--------+
| 22 | Manoj     | Kumar     |   5000 |
+----+-----------+-----------+--------+
1 row in set (0.00 sec)


mysql> select * from CERTIFICATE;
+----+------------------+
| id | certificate_name |
+----+------------------+
|  4 | MBA              |
|  5 | PMP              |
|  6 | MCA              |
+----+------------------+
3 rows in set (0.00 sec)


mysql> select * from EMP_CERT;
+-------------+----------------+
| employee_id | certificate_id |
+-------------+----------------+
|          22 |              4 |
|          22 |              5 |
|          22 |              6 |
+-------------+----------------+
3 rows in set (0.00 sec)
mysql>
```

Component Mappings

It is very much possible that an Entity class can have a reference to another class as a member variable. If the referred class does not have its own life cycle and completely depends on the life cycle of the owning entity class, then the referred class hence therefore is called as the **Component class**.

The mapping of Collection of Components is also possible in a similar way just as the mapping of regular Collections with minor configuration differences. We will see these two mappings in detail with examples.

Mapping type	Description
Component Mappings	Mapping for a class having a reference to another class as a member variable.

Hibernate – Component Mappings

A **Component** mapping is a mapping for a class having a reference to another class as a member variable. We have seen such mapping while having two tables and using <set> element in the mapping file. Now we will use <component> element in the mapping file and a single table would be used to keep the attributes contained inside the class variable.

Define RDBMS Tables

Consider a situation where we need to store our employee records in EMPLOYEE table, which will have the following structure:

```
create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name  VARCHAR(20) default NULL,
    salary     INT  default NULL,
    PRIMARY KEY (id)
);
```

Further, assume each employee will have an address, so let us add address specific fields in the same table as follows:

```
create table EMPLOYEE (
    id INT NOT  NULL auto_increment,
    first_name  VARCHAR(20) default NULL,
    last_name   VARCHAR(20) default NULL,
    salary      INT  default NULL,
    street_name VARCHAR(40) default NULL,
    city_name   VARCHAR(40) default NULL,
    state_name  VARCHAR(40) default NULL,
    zipcode     VARCHAR(10) default NULL,
    PRIMARY KEY (id)
);
```

Define POJO Classes

Let us implement our POJO class **Employee,** which will be used to persist the objects related to EMPLOYEE table.

```
import java.util.*;


public class Employee implements java.io.Serializable {
```

```
   private int id;
   private String firstName;
   private String lastName;
   private int salary;
   private Address address;

   public Employee() {}
   public Employee(String fname, String lname,
                   int salary, Address address ) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
      this.address = address;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
}
```

```
   public void setSalary( int salary ) {

      this.salary = salary;

   }


   public Address getAddress() {

      return address;

   }

   public void setAddress( Address address ) {

      this.address = address;

   }

}
```

We need to define another POJO class corresponding to ADDRESS entity having address related fields.

```
import java.util.*;


public class Address{

   private int id;

   private String street;

   private String city;

   private String state;

   private String zipcode;


   public Address() {}

   public Address(String street, String city,

                  String state, String zipcode) {

      this.street = street;

      this.city = city;

      this.state = state;

      this.zipcode = zipcode;

   }

   public int getId() {

      return id;

   }

   public void setId( int id ) {

      this.id = id;
```

```
   }

   public String getStreet() {

      return street;

   }

   public void setStreet( String street ) {

      this.street = street;

   }

   public String getCity() {

      return city;

   }

   public void setCity( String city ) {

      this.city = city;

   }

   public String getState() {

      return state;

   }

   public void setState( String state ) {

      this.state = state;

   }

   public String getZipcode() {

      return zipcode;

   }

   public void setZipcode( String zipcode ) {

      this.zipcode = zipcode;

   }


}
```

Define Hibernate Mapping File

Let us develop our mapping file, which instructs Hibernate how to map the defined classes to the database tables. The <component> element will be used to define the rule for all the fields associated with ADDRESS table.

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-mapping PUBLIC

  "-//Hibernate/Hibernate Mapping DTD//EN"
```

```
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>

   <class name="Employee" table="EMPLOYEE">

      <meta attribute="class-description">

         This class contains the employee detail.

      </meta>

      <id name="id" type="int" column="id">

         <generator class="native"/>

      </id>

      <component name="address" class="Address">

         <property name="street" column="street_name" type="string"/>

         <property name="city" column="city_name" type="string"/>

         <property name="state" column="state_name" type="string"/>

         <property name="zipcode" column="zipcode" type="string"/>

      </component>

      <property name="firstName" column="first_name" type="string"/>

      <property name="lastName" column="last_name" type="string"/>

      <property name="salary" column="salary" type="int"/>

   </class>


   <class name="Certificate" table="CERTIFICATE">

      <meta attribute="class-description">

         This class contains the certificate records.

      </meta>

      <id name="id" type="int" column="id">

         <generator class="native"/>

      </id>

      <property name="name" column="certificate_name" type="string"/>

   </class>


</hibernate-mapping>
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. You are already familiar with most of the mapping detail, but let us see all the elements of mapping file once again:

- The mapping document is an XML document having **<hibernate-mapping>** as the root element, which contains two <class> elements corresponding to each class.

- The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

- The **<meta>** element is optional element and can be used to create the class description.

- The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the id element is used to generate the primary key values automatically. The **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<component>** element sets the existence of different attributes of Address class inside Employee classes.

Create Application Class

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employee's records along with their certificates and then we will apply CRUD operations on those records.

```java
import java.util.*;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {

   private static SessionFactory factory;

   public static void main(String[] args) {
```

```
    try{
       factory = new Configuration().configure().buildSessionFactory();
    }catch (Throwable ex) {
       System.err.println("Failed to create sessionFactory object." + ex);
       throw new ExceptionInInitializerError(ex);
    }
    ManageEmployee ME = new ManageEmployee();


    /* Let us have one address object */
    Address address1 = ME.addAddress("Kondapur","Hyderabad","AP","532");


    /* Add employee records in the database */
    Integer empID1 = ME.addEmployee("Manoj", "Kumar", 4000, address1);


    /* Let us have another address object */
    Address address2 = ME.addAddress("Saharanpur","Ambehta","UP","111");

 /* Add another employee record in the database */
    Integer empID2 = ME.addEmployee("Dilip", "Kumar", 3000, address2);


    /* List down all the employees */
    ME.listEmployees();


    /* Update employee's salary records */
    ME.updateEmployee(empID1, 5000);


    /* List down all the employees */
    ME.listEmployees();


 }


 /* Method to add an address record in the database */
 public Address addAddress(String street, String city,
                           String state, String zipcode) {
    Session session = factory.openSession();
```

```
        Transaction tx = null;

        Integer addressID = null;

        Address address = null;

        try{

            tx = session.beginTransaction();

            address = new Address(street, city, state, zipcode);

            addressID = (Integer) session.save(address);

            tx.commit();

        }catch (HibernateException e) {

            if (tx!=null) tx.rollback();

            e.printStackTrace();

        }finally {

            session.close();

        }

        return address;

    }


    /* Method to add an employee record in the database */

    public Integer addEmployee(String fname, String lname,

                               int salary, Address address){

        Session session = factory.openSession();

        Transaction tx = null;

        Integer employeeID = null;

        try{

            tx = session.beginTransaction();

            Employee employee = new Employee(fname, lname, salary, address);

            employeeID = (Integer) session.save(employee);

            tx.commit();

        }catch (HibernateException e) {

            if (tx!=null) tx.rollback();

            e.printStackTrace();

        }finally {

            session.close();

        }

        return employeeID;
```

```
   }

   /* Method to list all the employees detail */
   public void listEmployees( ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         List employees = session.createQuery("FROM Employee").list();
         for (Iterator iterator =
                        employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
            System.out.println("  Salary: " + employee.getSalary());
            Address add = employee.getAddress();
            System.out.println("Address ");
            System.out.println("\tStreet: " +  add.getStreet());
            System.out.println("\tCity: " + add.getCity());
            System.out.println("\tState: " + add.getState());
            System.out.println("\tZipcode: " + add.getZipcode());
         }
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to update salary for an employee */
   public void updateEmployee(Integer EmployeeID, int salary ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
```

```
        tx = session.beginTransaction();

        Employee employee =

                  (Employee)session.get(Employee.class, EmployeeID);

        employee.setSalary( salary );

        session.update(employee);

        tx.commit();

     }catch (HibernateException e) {

        if (tx!=null) tx.rollback();

        e.printStackTrace();

     }finally {

        session.close();

     }

  }

}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure, you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result on the screen, and same time records would be created in EMPLOYEE table.

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Manoj  Last Name: Kumar  Salary: 4000

Address

       Street: Kondapur

       City: Hyderabad

       State: AP

       Zipcode: 532
```

```
First Name: Dilip   Last Name: Kumar   Salary: 3000

Address

        Street: Saharanpur

        City: Ambehta

        State: UP

        Zipcode: 111

First Name: Manoj   Last Name: Kumar   Salary: 5000

Address

        Street: Kondapur

        City: Hyderabad

        State: AP

        Zipcode: 532

First Name: Dilip   Last Name: Kumar   Salary: 3000

Address

        Street: Saharanpur

        City: Ambehta

        State: UP

        Zipcode: 111
```

If you check your EMPLOYEE table, it should have the following records:

```
mysql> select id, first_name,salary, street_name, state_name from EMPLOYEE;

+----+------------+--------+-------------+------------+
| id | first_name | salary | street_name | state_name |
+----+------------+--------+-------------+------------+
|  1 | Manoj      |   5000 | Kondapur    | AP         |
|  2 | Dilip      |   3000 | Saharanpur  | UP         |
+----+------------+--------+-------------+------------+
2 rows in set (0.00 sec)


mysql>
```

So far you have seen how Hibernate uses XML mapping file for the transformation of data from POJO to database tables and vice versa. Hibernate annotations are the newest way to define mappings without the use of XML file. You can use annotations in addition to or as a replacement of XML mapping metadata.

Hibernate Annotations is the powerful way to provide the metadata for the Object and Relational Table mapping. All the metadata is clubbed into the POJO java file along with the code, this helps the user to understand the table structure and POJO simultaneously during the development.

If you going to make your application portable to other EJB 3 compliant ORM applications, you must use annotations to represent the mapping information, but still if you want greater flexibility, then you should go with XML-based mappings.

Environment Setup for Hibernate Annotation

First of all you would have to make sure that you are using JDK 5.0 otherwise you need to upgrade your JDK to JDK 5.0 to take advantage of the native support for annotations.

Second, you will need to install the Hibernate 3.x annotations distribution package, available from the sourceforge: (Download Hibernate Annotation) and copy **hibernate-annotations.jar, lib/hibernate-comons-annotations.jar** and **lib/ejb3-persistence.jar** from the Hibernate Annotations distribution to your CLASSPATH.

Annotated Class Example

As I mentioned above while working with Hibernate Annotation, all the metadata is clubbed into the POJO java file along with the code, this helps the user to understand the table structure and POJO simultaneously during the development.

Consider we are going to use the following EMPLOYEE table to store our objects:

```
create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name  VARCHAR(20) default NULL,
    salary     INT  default NULL,
    PRIMARY KEY (id)
);
```

Following is the mapping of Employee class with annotations to map objects with the defined EMPLOYEE table:

```
import javax.persistence.*;
```

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
   @Id @GeneratedValue
   @Column(name = "id")
   private int id;

   @Column(name = "first_name")
   private String firstName;

   @Column(name = "last_name")
   private String lastName;

   @Column(name = "salary")
   private int salary;

   public Employee() {}
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
```

```
   }
   public int getSalary() {

      return salary;

   }
   public void setSalary( int salary ) {

      this.salary = salary;

   }
}
```

Hibernate detects that the @Id annotation is on a field and assumes that it should access properties of an object directly through fields at runtime. If you placed the @Id annotation on the getId() method, you would enable access to properties through getter and setter methods by default. Hence, all other annotations are also placed on either fields or getter methods, following the selected strategy.

Following section will explain the annotations used in the above class.

@Entity Annotation

The EJB 3 standard annotations are contained in the **javax.persistence** package, so we import this package as the first step. Second, we used the **@Entity** annotation to the Employee class, which marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

@Table Annotation

The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database.

The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table. For now, we are using just table name, which is EMPLOYEE.

@Id and @GeneratedValue Annotations

Each entity bean will have a primary key, which you annotate on the class with the **@Id** annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.

By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the **@GeneratedValue** annotation, which takes two parameters **strategy** and **generator** that I'm not going to discuss here, so let us use only the default key generation strategy. Letting Hibernate determine which generator type to use makes your code portable between different databases.

@Column Annotation

The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:

- **name** attribute permits the name of the column to be explicitly specified.

- **length** attribute permits the size of the column used to map a value particularly for a String value.

- **nullable** attribute permits the column to be marked NOT NULL when the schema is generated.

- **unique** attribute permits the column to be marked as containing only unique values.

Create Application Class

Finally, we will create our application class with the main() method to run the application. We will use this application to save few Employee's records and then we will apply CRUD operations on those records.

```java
import java.util.List;

import java.util.Date;

import java.util.Iterator;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.cfg.AnnotationConfiguration;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new AnnotationConfiguration().
                  configure().
                  //addPackage("com.xyz") //add package if used.
                  addAnnotatedClass(Employee.class).
                  buildSessionFactory();
```

```
    }catch (Throwable ex) {
        System.err.println("Failed to create sessionFactory object." + ex);
        throw new ExceptionInInitializerError(ex);
    }
    ManageEmployee ME = new ManageEmployee();


    /* Add few employee records in database */
    Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);
    Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
    Integer empID3 = ME.addEmployee("John", "Paul", 10000);


    /* List down all the employees */
    ME.listEmployees();


    /* Update employee's records */
    ME.updateEmployee(empID1, 5000);


    /* Delete an employee from the database */
    ME.deleteEmployee(empID2);


    /* List down new list of the employees */
    ME.listEmployees();
}
/* Method to CREATE an employee in the database */
public Integer addEmployee(String fname, String lname, int salary){
    Session session = factory.openSession();
    Transaction tx = null;
    Integer employeeID = null;
    try{
        tx = session.beginTransaction();
        Employee employee = new Employee();
        employee.setFirstName(fname);
        employee.setLastName(lname);
        employee.setSalary(salary);
        employeeID = (Integer) session.save(employee);
```

```
            tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
      return employeeID;
   }
   /* Method to  READ all the employees */
   public void listEmployees( ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         List employees = session.createQuery("FROM Employee").list();
         for (Iterator iterator =
                       employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
            System.out.println("  Salary: " + employee.getSalary());
         }
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to UPDATE salary for an employee */
   public void updateEmployee(Integer EmployeeID, int salary ){
      Session session = factory.openSession();
      Transaction tx = null;
```

```
        try{
            tx = session.beginTransaction();
            Employee employee =
                        (Employee)session.get(Employee.class, EmployeeID);
            employee.setSalary( salary );
                session.update(employee);
            tx.commit();
        }catch (HibernateException e) {
            if (tx!=null) tx.rollback();
            e.printStackTrace();
        }finally {
            session.close();
        }
    }
    /* Method to DELETE an employee from the records */
    public void deleteEmployee(Integer EmployeeID){
        Session session = factory.openSession();
        Transaction tx = null;
        try{
            tx = session.beginTransaction();
            Employee employee =
                        (Employee)session.get(Employee.class, EmployeeID);
            session.delete(employee);
            tx.commit();
        }catch (HibernateException e) {
            if (tx!=null) tx.rollback();
            e.printStackTrace();
        }finally {
            session.close();
        }
    }
}
```

Database Configuration

Now let us create **hibernate.cfg.xml** configuration file to define database related parameters.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">


<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>


   <!-- Assume students is the database name -->
   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/test
   </property>
   <property name="hibernate.connection.username">
      root
   </property>
   <property name="hibernate.connection.password">
      cohondob
   </property>


</session-factory>
</hibernate-configuration>
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure, you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Delete Employee.hbm.xml mapping file from the path.

- Create Employee.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result, and records would be created in EMPLOYEE table.

163

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Zara   Last Name: Ali   Salary: 1000

First Name: Daisy  Last Name: Das   Salary: 5000

First Name: John   Last Name: Paul  Salary: 10000

First Name: Zara   Last Name: Ali   Salary: 5000

First Name: John   Last Name: Paul  Salary: 10000
```

If you check your EMPLOYEE table, it should have the following records:

```
mysql> select * from EMPLOYEE;

+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
| 29 | Zara       | Ali       |   5000 |
| 31 | John       | Paul      |  10000 |
+----+------------+-----------+--------+
2 rows in set (0.00 sec
```

mysql>

13. HIBERNATE – QUERY LANGUAGE

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries, which in turns perform action on database.

Although you can use SQL statements directly with Hibernate using Native SQL, but I would recommend to use HQL whenever possible to avoid database portability hassles, and to take advantage of Hibernate's SQL generation and caching strategies.

Keywords like SELECT, FROM, and WHERE, etc., are not case sensitive, but properties like table and column names are case sensitive in HQL.

FROM Clause

You will use **FROM** clause if you want to load a complete persistent objects into memory. Following is the simple syntax of using FROM clause:

```
String hql = "FROM Employee";

Query query = session.createQuery(hql);

List results = query.list();
```

If you need to fully qualify a class name in HQL, just specify the package and class name as follows:

```
String hql = "FROM com.hibernatebook.criteria.Employee";

Query query = session.createQuery(hql);

List results = query.list();
```

AS Clause

The **AS** clause can be used to assign aliases to the classes in your HQL queries, especially when you have the long queries. For instance, our previous simple example would be the following:

```
String hql = "FROM Employee AS E";

Query query = session.createQuery(hql);

List results = query.list();
```

The **AS** keyword is optional and you can also specify the alias directly after the class name, as follows:

```
String hql = "FROM Employee E";
```

```
Query query = session.createQuery(hql);

List results = query.list();
```

SELECT Clause

The **SELECT** clause provides more control over the result set then the from clause. If you want to obtain few properties of objects instead of the complete object, use the SELECT clause. Following is the simple syntax of using SELECT clause to get just first_name field of the Employee object:

```
String hql = "SELECT E.firstName FROM Employee E";

Query query = session.createQuery(hql);

List results = query.list();
```

It is notable here that **Employee.firstName** is a property of Employee object rather than a field of the EMPLOYEE table.

WHERE Clause

If you want to narrow the specific objects that are returned from storage, you use the WHERE clause. Following is the simple syntax of using WHERE clause:

```
String hql = "FROM Employee E WHERE E.id = 10";

Query query = session.createQuery(hql);

List results = query.list();
```

ORDER BY Clause

To sort your HQL query's results, you will need to use the **ORDER BY** clause. You can order the results by any property on the objects in the result set either ascending (ASC) or descending (DESC). Following is the simple syntax of using ORDER BY clause:

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";

Query query = session.createQuery(hql);

List results = query.list();
```

If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas as follows:

```
String hql = "FROM Employee E WHERE E.id > 10 " +
             "ORDER BY E.firstName DESC, E.salary DESC ";

Query query = session.createQuery(hql);

List results = query.list();
```

GROUP by Clause

This clause lets Hibernate pull information from the database and group it based on a value of an attribute and, typically, use the result to include an aggregate value. Following is the simple syntax of using GROUP BY clause:

```
String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " +
             "GROUP BY E.firstName";
Query query = session.createQuery(hql);
List results = query.list();
```

Using Named Parameters

Hibernate supports named parameters in its HQL queries. This makes writing HQL queries that accept input from the user easy and you do not have to defend against SQL injection attacks. Following is the simple syntax of using named parameters:

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id",10);
List results = query.list();
```

UPDATE Clause

Bulk updates are new to HQL with Hibernate 3, and delete work differently in Hibernate 3 than they did in Hibernate 2. The Query interface now contains a method called executeUpdate() for executing HQL UPDATE or DELETE statements.

The **UPDATE** clause can be used to update one or more properties of an one or more objects. Following is the simple syntax of using UPDATE clause:

```
String hql = "UPDATE Employee set salary = :salary "  +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

DELETE Clause

The **DELETE** clause can be used to delete one or more objects. Following is the simple syntax of using DELETE clause:

```
String hql = "DELETE FROM Employee "  +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

INSERT Clause

HQL supports **INSERT INTO** clause only where records can be inserted from one object to another object. Following is the simple syntax of using INSERT INTO clause:

```
String hql = "INSERT INTO Employee(firstName, lastName, salary)"  +
             "SELECT firstName, lastName, salary FROM old_employee";
Query query = session.createQuery(hql);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

Aggregate Methods

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL and following is the list of the available functions:

S.N.	Functions	Description
1	avg(property name)	The average of a property's value
2	count(property name or *)	The number of times a property occurs in the results
3	max(property name)	The maximum value of the property values
4	min(property name)	The minimum value of the property values
5	sum(property name)	The sum total of the property values

The **distinct** keyword only counts the unique values in the row set. The following query will return only unique count:

```
String hql = "SELECT count(distinct E.firstName) FROM Employee E";
Query query = session.createQuery(hql);
```

```
List results = query.list();
```

Pagination using Query

There are two methods of the Query interface for pagination.

S.N.	Method & Description
1	**Query setFirstResult(int startPosition)** This method takes an integer that represents the first row in your result set, starting with row 0.
2	**Query setMaxResults(int maxResult)** This method tells Hibernate to retrieve a fixed number **maxResults** of objects.

Using above two methods together, we can construct a paging component in our web or Swing application. Following is the example, which you can extend to fetch 10 rows at a time:

```
String hql = "FROM Employee";

Query query = session.createQuery(hql);

query.setFirstResult(1);

query.setMaxResults(10);

List results = query.list();
```

14. HIBERNATE – CRITERIA QUERIES

Hibernate provides alternate ways of manipulating objects and in turn data available in RDBMS tables. One of the methods is Criteria API, which allows you to build up a criteria query object programmatically where you can apply filtration rules and logical conditions.

The Hibernate **Session** interface provides **createCriteria()** method, which can be used to create a **Criteria** object that returns instances of the persistence object's class when your application executes a criteria query.

Following is the simplest example of a criteria query is one, which will simply return every object that corresponds to the Employee class.

```
Criteria cr = session.createCriteria(Employee.class);
List results = cr.list();
```

Restrictions with Criteria

You can use **add()** method available for **Criteria** object to add restriction for a criteria query. Following is the example to add a restriction to return the records with salary is equal to 2000:

```
Criteria cr = session.createCriteria(Employee.class);
cr.add(Restrictions.eq("salary", 2000));
List results = cr.list();
```

Following are the few more examples covering different scenarios and can be used as per the requirement:

```
Criteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));

// To get records having salary less than 2000
cr.add(Restrictions.lt("salary", 2000));

// To get records having fistName starting with zara
cr.add(Restrictions.like("firstName", "zara%"));

// Case sensitive form of the above restriction.
```

```
cr.add(Restrictions.ilike("firstName", "zara%"));


// To get records having salary in between 1000 and 2000

cr.add(Restrictions.between("salary", 1000, 2000));


// To check if the given property is null

cr.add(Restrictions.isNull("salary"));


// To check if the given property is not null

cr.add(Restrictions.isNotNull("salary"));


// To check if the given property is empty

cr.add(Restrictions.isEmpty("salary"));


// To check if the given property is not empty

cr.add(Restrictions.isNotEmpty("salary"));
```

You can create AND or OR conditions using LogicalExpression restrictions as follows:

```
Criteria cr = session.createCriteria(Employee.class);


Criterion salary = Restrictions.gt("salary", 2000);

Criterion name = Restrictions.ilike("firstNname","zara%");


// To get records matching with OR condistions

LogicalExpression orExp = Restrictions.or(salary, name);

cr.add( orExp );

// To get records matching with AND condistions

LogicalExpression andExp = Restrictions.and(salary, name);

cr.add( andExp );

List results = cr.list();
```

Pagination Using Criteria

There are two methods of the Criteria interface for pagination.

S.N.	Method & Description
1	**public Criteria setFirstResult(int firstResult)** This method takes an integer that represents the first row in your result set, starting with row 0.
2	**public Criteria setMaxResults(int maxResults)** This method tells Hibernate to retrieve a fixed number **maxResults** of objects.

Using above two methods together, we can construct a paging component in our web or Swing application. Following is the example, which you can extend to fetch 10 rows at a time:

```
Criteria cr = session.createCriteria(Employee.class);

cr.setFirstResult(1);

cr.setMaxResults(10);

List results = cr.list();
```

Sorting the Results

The Criteria API provides the **org.hibernate.criterion.Order** class to sort your result set in either ascending or descending order, according to one of your object's properties. This example demonstrates how you would use the Order class to sort the result set:

```
Criteria cr = session.createCriteria(Employee.class);
// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));


// To sort records in descening order
crit.addOrder(Order.desc("salary"));


// To sort records in ascending order
crit.addOrder(Order.asc("salary"));
List results = cr.list();
```

Projections & Aggregations

The Criteria API provides the **org.hibernate.criterion.Projections** class, which can be used to get average, maximum, or minimum of the property values. The Projections class

is similar to the Restrictions class, in that it provides several static factory methods for obtaining **Projection** instances.

Following are the few examples covering different scenarios and can be used as per requirement:

```
Criteria cr = session.createCriteria(Employee.class);


// To get total row count.

cr.setProjection(Projections.rowCount());


// To get average of a property.

cr.setProjection(Projections.avg("salary"));


// To get distinct count of a property.

cr.setProjection(Projections.countDistinct("firstName"));


// To get maximum of a property.

cr.setProjection(Projections.max("salary"));


// To get minimum of a property.

cr.setProjection(Projections.min("salary"));


// To get sum of a property.

cr.setProjection(Projections.sum("salary"));
```

Criteria Queries Example

Consider the following POJO class:

```
public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private int salary;

    public Employee() {}
    public Employee(String fname, String lname, int salary) {
        this.firstName = fname;
```

```
      this.lastName = lname;

      this.salary = salary;

   }

   public int getId() {

      return id;

   }

   public void setId( int id ) {

      this.id = id;

   }

   public String getFirstName() {

      return firstName;

   }

   public void setFirstName( String first_name ) {

      this.firstName = first_name;

   }

   public String getLastName() {

      return lastName;

   }

   public void setLastName( String last_name ) {

      this.lastName = last_name;

   }

   public int getSalary() {

      return salary;

   }

   public void setSalary( int salary ) {

      this.salary = salary;

   }

}
```

Let us create the following EMPLOYEE table to store Employee objects:

```
create table EMPLOYEE (

   id INT NOT NULL auto_increment,

   first_name VARCHAR(20) default NULL,

   last_name  VARCHAR(20) default NULL,

   salary     INT  default NULL,

   PRIMARY KEY (id)
```

```
);
```

Following will be the mapping file.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>
    <class name="Employee" table="EMPLOYEE">
        <meta attribute="class-description">
            This class contains the employee detail.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="native"/>
        </id>
        <property name="firstName" column="first_name" type="string"/>
        <property name="lastName" column="last_name" type="string"/>
        <property name="salary" column="salary" type="int"/>
    </class>
</hibernate-mapping>
```

Finally, we will create our application class with the main() method to run the application where we will use **Criteria** queries:

```java
import java.util.List;
import java.util.Date;
import java.util.Iterator;


import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.Criteria;
import org.hibernate.criterion.Restrictions;
import org.hibernate.criterion.Projections;
import org.hibernate.cfg.Configuration;
```

```
public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {
        try{
            factory = new Configuration().configure().buildSessionFactory();
        }catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
        ManageEmployee ME = new ManageEmployee();

        /* Add few employee records in database */
        Integer empID1 = ME.addEmployee("Zara", "Ali", 2000);
        Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
        Integer empID3 = ME.addEmployee("John", "Paul", 5000);
        Integer empID4 = ME.addEmployee("Mohd", "Yasee", 3000);

        /* List down all the employees */
        ME.listEmployees();

        /* Print Total employee's count */
        ME.countEmployee();

        /* Print Toatl salary */
        ME.totalSalary();
    }
    /* Method to CREATE an employee in the database */
    public Integer addEmployee(String fname, String lname, int salary){
        Session session = factory.openSession();
        Transaction tx = null;
        Integer employeeID = null;
        try{
            tx = session.beginTransaction();
            Employee employee = new Employee(fname, lname, salary);
```

```
        employeeID = (Integer) session.save(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
    return employeeID;
}


/* Method to  READ all the employees having salary more than 2000 */
public void listEmployees( ){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        Criteria cr = session.createCriteria(Employee.class);
        // Add restriction.
        cr.add(Restrictions.gt("salary", 2000));
        List employees = cr.list();

        for (Iterator iterator =
                        employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
            System.out.println("  Salary: " + employee.getSalary());
        }
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
```

```
      }
   }
   /* Method to print total number of records */
   public void countEmployee(){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Criteria cr = session.createCriteria(Employee.class);

         // To get total row count.
         cr.setProjection(Projections.rowCount());
         List rowCount = cr.list();

         System.out.println("Total Coint: " + rowCount.get(0) );
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
  /* Method to print sum of salaries */
   public void totalSalary(){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Criteria cr = session.createCriteria(Employee.class);

         // To get total salary.
         cr.setProjection(Projections.sum("salary"));
         List totalSalary = cr.list();
```

```
        System.out.println("Total Salary: " + totalSalary.get(0) );

        tx.commit();

    }catch (HibernateException e) {

        if (tx!=null) tx.rollback();

        e.printStackTrace();

    }finally {

        session.close();

    }

  }

}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure, you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result, and records would be created in the EMPLOYEE table.

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Daisy  Last Name: Das  Salary: 5000

First Name: John  Last Name: Paul  Salary: 5000

First Name: Mohd  Last Name: Yasee  Salary: 3000

Total Coint: 4

Total Salary: 15000
```

If you check your EMPLOYEE table, it should have the following records:

```
mysql> select * from EMPLOYEE;

+----+------------+-----------+--------+

| id | first_name | last_name | salary |
```

```
+----+-----------+-----------+--------+
| 14 | Zara      | Ali       |   2000 |
| 15 | Daisy     | Das       |   5000 |
| 16 | John      | Paul      |   5000 |
| 17 | Mohd      | Yasee     |   3000 |
+----+-----------+-----------+--------+
4 rows in set (0.00 sec)
mysql>
```

15. HIBERNATE – NATIVE SQL

You can use native SQL to express database queries if you want to utilize database-specific features such as query hints or the CONNECT keyword in Oracle. Hibernate 3.x allows you to specify handwritten SQL, including stored procedures, for all create, update, delete, and load operations.

Your application will create a native SQL query from the session with the **createSQLQuery()** method on the Session interface.:

```
public SQLQuery createSQLQuery(String sqlString) throws HibernateException
```

After you pass a string containing the SQL query to the createSQLQuery() method, you can associate the SQL result with either an existing Hibernate entity, a join, or a scalar result using addEntity(), addJoin(), and addScalar() methods respectively.

Scalar Queries

The most basic SQL query is to get a list of scalars (values) from one or more tables. Following is the syntax for using native SQL for scalar values:

```
String sql = "SELECT first_name, salary FROM EMPLOYEE";

SQLQuery query = session.createSQLQuery(sql);

query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);

List results = query.list();
```

Entity Queries

The above queries were all about returning scalar values, basically returning the "raw" values from the result set. Following is the syntax to get entity objects as a whole from a native sql query via addEntity().

```
String sql = "SELECT * FROM EMPLOYEE";

SQLQuery query = session.createSQLQuery(sql);

query.addEntity(Employee.class);

List results = query.list();
```

Named SQL Queries

Following is the syntax to get entity objects from a native sql query via addEntity() and using named SQL query.

```
String sql = "SELECT * FROM EMPLOYEE WHERE id = :employee_id";
```

```
SQLQuery query = session.createSQLQuery(sql);

query.addEntity(Employee.class);

query.setParameter("employee_id", 10);

List results = query.list();
```

Native SQL Example

Consider the following POJO class:

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
}
```

```
    public void setLastName( String last_name ) {

        this.lastName = last_name;

    }

    public int getSalary() {

        return salary;

    }

    public void setSalary( int salary ) {

        this.salary = salary;

    }

}
```

Let us create the following EMPLOYEE table to store Employee objects:

```
create table EMPLOYEE (

    id INT NOT NULL auto_increment,

    first_name VARCHAR(20) default NULL,

    last_name  VARCHAR(20) default NULL,

    salary     INT  default NULL,

    PRIMARY KEY (id)

);
```

Following will be the mapping file:

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-mapping PUBLIC

 "-//Hibernate/Hibernate Mapping DTD//EN"

 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>

    <class name="Employee" table="EMPLOYEE">

        <meta attribute="class-description">

            This class contains the employee detail.

        </meta>

        <id name="id" type="int" column="id">

            <generator class="native"/>

        </id>

        <property name="firstName" column="first_name" type="string"/>

        <property name="lastName" column="last_name" type="string"/>
```

```
        <property name="salary" column="salary" type="int"/>

    </class>

</hibernate-mapping>
```

Finally, we will create our application class with the main() method to run the application where we will use **Native SQL** queries:

```
import java.util.*;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.SQLQuery;

import org.hibernate.Criteria;

import org.hibernate.Hibernate;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {
        try{
            factory = new Configuration().configure().buildSessionFactory();
        }catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
        ManageEmployee ME = new ManageEmployee();


        /* Add few employee records in database */
        Integer empID1 = ME.addEmployee("Zara", "Ali", 2000);
        Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
        Integer empID3 = ME.addEmployee("John", "Paul", 5000);
        Integer empID4 = ME.addEmployee("Mohd", "Yasee", 3000);


        /* List down employees and their salary using Scalar Query */
        ME.listEmployeesScalar();
```

```
    /* List down complete employees information using Entity Query */

    ME.listEmployeesEntity();

}

/* Method to CREATE an employee in the database */

public Integer addEmployee(String fname, String lname, int salary){

    Session session = factory.openSession();

    Transaction tx = null;

    Integer employeeID = null;

    try{

       tx = session.beginTransaction();

       Employee employee = new Employee(fname, lname, salary);

       employeeID = (Integer) session.save(employee);

       tx.commit();

    }catch (HibernateException e) {

       if (tx!=null) tx.rollback();

       e.printStackTrace();

    }finally {

       session.close();

    }

    return employeeID;

}


/* Method to  READ all the employees using Scalar Query */

public void listEmployeesScalar( ){

    Session session = factory.openSession();

    Transaction tx = null;

    try{

       tx = session.beginTransaction();

       String sql = "SELECT first_name, salary FROM EMPLOYEE";

       SQLQuery query = session.createSQLQuery(sql);

       query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);

       List data = query.list();


       for(Object object : data)
```

```
        {
            Map row = (Map)object;

            System.out.print("First Name: " + row.get("first_name"));

            System.out.println(", Salary: " + row.get("salary"));

        }
        tx.commit();

    }catch (HibernateException e) {

        if (tx!=null) tx.rollback();

        e.printStackTrace();

    }finally {

        session.close();

    }

}


/* Method to  READ all the employees using Entity Query */
public void listEmployeesEntity( ){
    Session session = factory.openSession();

    Transaction tx = null;

    try{
        tx = session.beginTransaction();

        String sql = "SELECT * FROM EMPLOYEE";

        SQLQuery query = session.createSQLQuery(sql);

        query.addEntity(Employee.class);

        List employees = query.list();


        for (Iterator iterator =
                        employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();

            System.out.print("First Name: " + employee.getFirstName());

            System.out.print("  Last Name: " + employee.getLastName());

            System.out.println("  Salary: " + employee.getSalary());

        }
        tx.commit();

    }catch (HibernateException e) {

        if (tx!=null) tx.rollback();
```

```
        e.printStackTrace();
    }finally {
        session.close();
    }
}
}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure, you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program.

You would get the following result, and records would be created in the EMPLOYEE table.

```
$java ManageEmployee

.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


First Name: Zara, Salary: 2000

First Name: Daisy, Salary: 5000

First Name: John, Salary: 5000

First Name: Mohd, Salary: 3000

First Name: Zara  Last Name: Ali  Salary: 2000

First Name: Daisy  Last Name: Das  Salary: 5000

First Name: John  Last Name: Paul  Salary: 5000

First Name: Mohd  Last Name: Yasee  Salary: 3000
```

If you check your EMPLOYEE table, it should have the following records:

```
mysql> select * from EMPLOYEE;

+----+------------+-----------+--------+

| id | first_name | last_name | salary |

+----+------------+-----------+--------+
```

```
| 26 | Zara       | Ali        |   2000 |
| 27 | Daisy      | Das        |   5000 |
| 28 | John       | Paul       |   5000 |
| 29 | Mohd       | Yasee      |   3000 |
+----+------------+------------+--------+
4 rows in set (0.00 sec)

mysql>
```

16. HIBERNATE – CACHING

Caching is a mechanism to enhance the performance of a system. It is a buffer memory that lies between the application and the database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.

Caching is important to Hibernate as well. It utilizes a multilevel caching scheme as explained below:
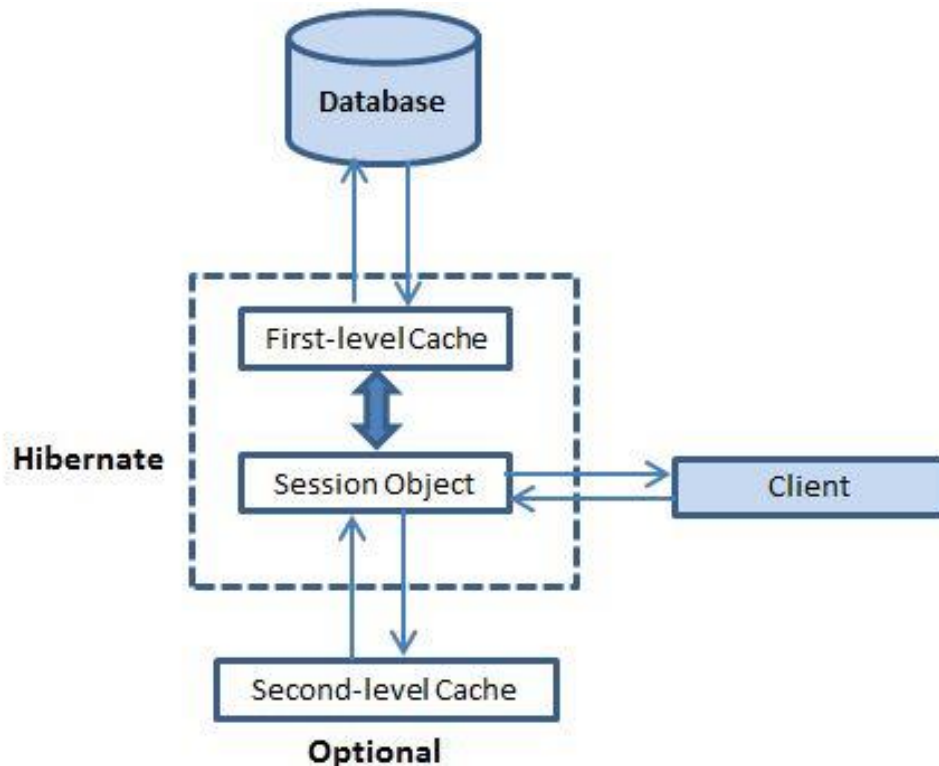
First-level Cache

The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.

If you issue multiple updates to an object, Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.

Second-level Cache

Second level cache is an optional cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second-

level cache can be configured on a per-class and per-collection basis and mainly responsible for caching objects across sessions.

Any third-party cache can be used with Hibernate. An **org.hibernate.cache.CacheProvider** interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation.

Query-level Cache

Hibernate also implements a cache for query resultsets that integrates closely with the second-level cache.

This is an optional feature and requires two additional physical cache regions that hold the cached query results and the timestamps when a table was last updated. This is only useful for queries that are run frequently with the same parameters.

The Second Level Cache

Hibernate uses first-level cache by default and you have nothing to do to use first-level cache. Let's go straight to the optional second-level cache. Not all classes benefit from caching, so it's important to be able to disable the second-level cache.

The Hibernate second-level cache is set up in two steps. First, you have to decide which concurrency strategy to use. After that, you configure cache expiration and physical cache attributes using the cache provider.

Concurrency Strategies

A concurrency strategy is a mediator, which is responsible for storing items of data in the cache and retrieving them from the cache. If you are going to enable a second-level cache, you will have to decide, for each persistent class and collection, which cache concurrency strategy to use.

- **Transactional:** Use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.

- **Read-write:** Again use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.

- **Nonstrict-read-write:** This strategy makes no guarantee of consistency between the cache and the database. Use this strategy if data hardly ever changes and a small likelihood of stale data is not of critical concern.

- **Read-only:** A concurrency strategy suitable for data, which never changes. Use it for reference data only.

- If we are going to use second-level caching for our **Employee** class, let us add the mapping element required to tell Hibernate to cache Employee instances using read-write strategy.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
```

```
  "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">


<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <cache usage="read-write"/>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>
</hibernate-mapping>
```

The usage="read-write" attribute tells Hibernate to use a read-write concurrency strategy for the defined cache.

Cache Provider

Your next step after considering the concurrency strategies, you will use your cache candidate classes to pick a cache provider. Hibernate forces you to choose a single cache provider for the whole application.

S.N.	Cache Name	Description
1	EHCache	It can cache in memory or on disk and clustered caching and it supports the optional Hibernate query result cache.
2	OSCache	Supports caching to memory and disk in a single JVM with a rich set of expiration policies and query cache support.
3	warmCache	A cluster cache based on JGroups. It uses clustered invalidation, but doesn't support the Hibernate query cache.
4	JBoss Cache	A fully transactional replicated clustered cache also based on the JGroups multicast library. It supports replication or invalidation, synchronous or asynchronous communication, and optimistic and pessimistic locking. The Hibernate query cache is supported.

Every cache provider is not compatible with every concurrency strategy. The following compatibility matrix will help you choose an appropriate combination.

Strategy/Provider	Read-only	Nonstrictread-write	Read-write	Transactional
EHCache	X	X	X	
OSCache	X	X	X	
SwarmCache	X	X		
JBoss Cache	X			X

You will specify a cache provider in hibernate.cfg.xml configuration file. We choose EHCache as our second-level cache provider:

```xml
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-configuration SYSTEM

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">


<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>


   <!-- Assume students is the database name -->
   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/test
   </property>
   <property name="hibernate.connection.username">
      root
   </property>
   <property name="hibernate.connection.password">
      root123
   </property>
   <property name="hibernate.cache.provider_class">
      org.hibernate.cache.EhCacheProvider
   </property>
```

```
    <!-- List of XML mapping files -->

    <mapping resource="Employee.hbm.xml"/>


</session-factory>

</hibernate-configuration>
```

Now, you need to specify the properties of the cache regions. EHCache has its own configuration file, **ehcache.xml**, which should be in the CLASSPATH of the application. A cache configuration in ehcache.xml for the Employee class may look like this:

```
<diskStore path="java.io.tmpdir"/>

<defaultCache

maxElementsInMemory="1000"

eternal="false"

timeToIdleSeconds="120"

timeToLiveSeconds="120"

overflowToDisk="true"

/>


<cache name="Employee"

maxElementsInMemory="500"

eternal="true"

timeToIdleSeconds="0"

timeToLiveSeconds="0"

overflowToDisk="false"

/>
```

That's it, now we have second-level caching enabled for the Employee class and Hibernate, now hits the second-level cache whenever you navigate to an Employee or when you load an Employee by identifier.

You should analyze your all the classes and choose appropriate caching strategy for each of the classes. Sometime, second-level caching may downgrade the performance of the application. So, it is recommended to benchmark your application first, without enabling caching and later on enable your well suited caching and check the performance. If caching is not improving system performance, then there is no point in enabling any type of caching.

The Query-level Cache

To use the query cache, you must first activate it using the **hibernate.cache.use_query_cache="true"** property in the configuration file. By

193

setting this property to true, you make Hibernate create the necessary caches in memory to hold the query and identifier sets.

Next, to use the query cache, you use the setCacheable(Boolean) method of the Query class. For example:

```
Session session = SessionFactory.openSession();

Query query = session.createQuery("FROM EMPLOYEE");

query.setCacheable(true);

List users = query.list();

SessionFactory.closeSession();
```

Hibernate also supports very fine-grained cache support through the concept of a cache region. A cache region is part of the cache that's given a name.

```
Session session = SessionFactory.openSession();

Query query = session.createQuery("FROM EMPLOYEE");

query.setCacheable(true);

query.setCacheRegion("employee");

List users = query.list();

SessionFactory.closeSession();
```

This code uses the method to tell Hibernate to store and look for the query in the employee area of the cache.

Consider a situation when you need to upload a large number of records into your database using Hibernate. Following is the code snippet to achieve this using Hibernate:

```
Session session = SessionFactory.openSession();

Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {

    Employee employee = new Employee(.....);

    session.save(employee);

}
tx.commit();

session.close();
```

By default, Hibernate will cache all the persisted objects in the session-level cache and ultimately your application would fall over with an **OutOfMemoryException** somewhere around the 50,000th row. You can resolve this problem, if you are using **batch processing** with Hibernate.

To use the batch processing feature, first set **hibernate.jdbc.batch_size** as batch size to a number either at 20 or 50 depending on object size. This will tell the hibernate container that every X rows to be inserted as batch. To implement this in your code, we would need to do little modification as follows:

```
Session session = SessionFactory.openSession();

Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {

    Employee employee = new Employee(.....);

    session.save(employee);

     if( i % 50 == 0 ) { // Same as the JDBC batch size

        //flush a batch of inserts and release memory:

        session.flush();

        session.clear();

    }

}
tx.commit();

session.close();
```

Above code will work fine for the INSERT operation, but if you are willing to make UPDATE operation, then you can achieve using the following code:

```
Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();


ScrollableResults employeeCursor = session.createQuery("FROM EMPLOYEE")
                                    .scroll();

int count = 0;


while ( employeeCursor.next() ) {

    Employee employee = (Employee) employeeCursor.get(0);

    employee.updateEmployee();

    seession.update(employee);

    if ( ++count % 50 == 0 ) {

        session.flush();

        session.clear();

    }

}

tx.commit();

session.close();
```

Batch Processing Example

Let us modify the configuration file to add **hibernate.jdbc.batch_size** property:

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-configuration SYSTEM

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">


<hibernate-configuration>

    <session-factory>

    <property name="hibernate.dialect">

        org.hibernate.dialect.MySQLDialect

    </property>

    <property name="hibernate.connection.driver_class">

        com.mysql.jdbc.Driver

    </property>


    <!-- Assume students is the database name -->
```

```
    <property name="hibernate.connection.url">

        jdbc:mysql://localhost/test

    </property>

    <property name="hibernate.connection.username">

        root

    </property>

    <property name="hibernate.connection.password">

        root123

    </property>

    <property name="hibernate.jdbc.batch_size">

        50

    </property>


    <!-- List of XML mapping files -->

    <mapping resource="Employee.hbm.xml"/>


</session-factory>
</hibernate-configuration>
```

Consider the following POJO Employee class:

```
public class Employee {

    private int id;

    private String firstName;

    private String lastName;

    private int salary;


    public Employee() {}

    public Employee(String fname, String lname, int salary) {

        this.firstName = fname;

        this.lastName = lname;

        this.salary = salary;

    }

    public int getId() {

        return id;

    }

    public void setId( int id ) {
```

```
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName( String first_name ) {
        this.firstName = first_name;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName( String last_name ) {
        this.lastName = last_name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary( int salary ) {
        this.salary = salary;
    }
}
```

Let us create the following EMPLOYEE table to store the Employee objects:

```
create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name  VARCHAR(20) default NULL,
    salary     INT  default NULL,
    PRIMARY KEY (id)
);
```

Following will be the mapping file to map the Employee objects with EMPLOYEE table:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping>

    <class name="Employee" table="EMPLOYEE">

        <meta attribute="class-description">

            This class contains the employee detail.

        </meta>

        <id name="id" type="int" column="id">

            <generator class="native"/>

        </id>

        <property name="firstName" column="first_name" type="string"/>

        <property name="lastName" column="last_name" type="string"/>

        <property name="salary" column="salary" type="int"/>

    </class>

</hibernate-mapping>
```

Finally, we will create our application class with the main() method to run the application where we will use **flush()** and **clear()** methods available with Session object so that Hibernate keeps writing these records into the database instead of caching them in the memory.

```
import java.util.*;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {
        try{
            factory = new Configuration().configure().buildSessionFactory();
        }catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }

        ManageEmployee ME = new ManageEmployee();
```

```
      /* Add employee records in batches */

      ME.addEmployees( );

   }

   /* Method to create employee records in batches */

   public void addEmployees( ){

      Session session = factory.openSession();

      Transaction tx = null;

      Integer employeeID = null;

      try{

         tx = session.beginTransaction();

         for ( int i=0; i<100000; i++ ) {

            String fname = "First Name " + i;

            String lname = "Last Name " + i;

            Integer salary = i;

            Employee employee = new Employee(fname, lname, salary);

            session.save(employee);

            if( i % 50 == 0 ) {

               session.flush();

               session.clear();

            }

         }

         tx.commit();

      }catch (HibernateException e) {

         if (tx!=null) tx.rollback();

         e.printStackTrace();

      }finally {

         session.close();

      }

      return ;

   }

}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure, you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained above.

- Create Employee.hbm.xml mapping file as shown above.

- Create Employee.java source file as shown above and compile it.

- Create ManageEmployee.java source file as shown above and compile it.

- Execute ManageEmployee binary to run the program, which will create 100000 records in EMPLOYEE table.

As you have learnt that in Hibernate, an object will be created and persisted. Once the object has been changed, it must be saved back to the database. This process continues until the next time the object is needed, and it will be loaded from the persistent store.

Thus an object passes through different stages in its life cycle and **Interceptor Interface** provides methods, which can be called at different stages to perform some required tasks. These methods are callbacks from the session to the application, allowing the application to inspect and/or manipulate properties of a persistent object before it is saved, updated, deleted or loaded. Following is the list of all the methods available within the Interceptor interface:

S.N.	Method and Description
1	**findDirty()** This method is be called when the **flush()** method is called on a Session object.
2	**instantiate()** This method is called when a persisted class is instantiated.
3	**isUnsaved()** This method is called when an object is passed to the **saveOrUpdate()** method/
4	**onDelete()** This method is called before an object is deleted.
5	**onFlushDirty()** This method is called when Hibernate detects that an object is dirty (i.e. have been changed) during a flush i.e. update operation.
6	**onLoad()** This method is called before an object is initialized.
7	**onSave()** This method is called before an object is saved.
8	**postFlush()** This method is called after a flush has occurred and an object has been updated in memory.
9	**preFlush()** This method is called before a flush.

Hibernate Interceptor gives us total control over how an object will look to both the application and the database.

How to Use Interceptors?

To build an interceptor, you can either implement **Interceptor** class directly or extend **EmptyInterceptor** class. Following will be the simple steps to use Hibernate Interceptor functionality.

202

Create Interceptors

We will extend EmptyInterceptor in our example where Interceptor's method will be called automatically when **Employee** object is created and updated. You can implement more methods as per your requirements.

```
import java.io.Serializable;

import java.util.Date;

import java.util.Iterator;


import org.hibernate.EmptyInterceptor;

import org.hibernate.Transaction;

import org.hibernate.type.Type;
public class MyInterceptor extends EmptyInterceptor {
    private int updates;

    private int creates;

    private int loads;


    public void onDelete(Object entity,
                    Serializable id,
                    Object[] state,
                    String[] propertyNames,
                    Type[] types) {
        // do nothing

    }


    // This method is called when Employee object gets updated.
    public boolean onFlushDirty(Object entity,
                    Serializable id,
                    Object[] currentState,
                    Object[] previousState,
                    String[] propertyNames,
                    Type[] types) {
        if ( entity instanceof Employee ) {
            System.out.println("Update Operation");
            return true;
        }
        return false;
```

```
    }
    public boolean onLoad(Object entity,
                    Serializable id,
                    Object[] state,
                    String[] propertyNames,
                    Type[] types) {
        // do nothing
        return true;
    }
    // This method is called when Employee object gets created.
    public boolean onSave(Object entity,
                    Serializable id,
                    Object[] state,
                    String[] propertyNames,
                    Type[] types) {
        if ( entity instanceof Employee ) {
            System.out.println("Create Operation");
            return true;
        }
        return false;
    }
    //called before commit into database
    public void preFlush(Iterator iterator) {
        System.out.println("preFlush");
    }
    //called after committed into database
    public void postFlush(Iterator iterator) {
        System.out.println("postFlush");
    }
}
```

Create POJO Classes

Now, let us modify a little bit our first example where we used EMPLOYEE table and Employee class to play with:

```
public class Employee {
```

```
   private int id;
   private String firstName;
   private String lastName;
   private int salary;


   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
      this.salary = salary;
   }
```

```
}
```

Create Database Tables

Second step would be creating tables in your database. There would be one table corresponding to each object, you are willing to provide persistence. Consider the objects explained above, need to be stored and retrieved into the following RDBMS table:

```
create table EMPLOYEE (
    id INT NOT NULL auto_increment,
    first_name VARCHAR(20) default NULL,
    last_name  VARCHAR(20) default NULL,
    salary     INT  default NULL,
    PRIMARY KEY (id)
);
```

Create Mapping Configuration File

This step is to create a mapping file that instructs Hibernate — how to map the defined class or classes to the database tables.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="Employee" table="EMPLOYEE">
        <meta attribute="class-description">
            This class contains the employee detail.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="native"/>
        </id>
        <property name="firstName" column="first_name" type="string"/>
        <property name="lastName" column="last_name" type="string"/>
        <property name="salary" column="salary" type="int"/>
    </class>
</hibernate-mapping>
```

Create Application Class

Finally, we will create our application class with the main() method to run the application. Here, it should be noted that while creating session object, we used our Interceptor class as an argument.

```java
import java.util.List;

import java.util.Date;

import java.util.Iterator;


import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;


public class ManageEmployee {

    private static SessionFactory factory;

    public static void main(String[] args) {

        try{

            factory = new Configuration().configure().buildSessionFactory();

        }catch (Throwable ex) {

            System.err.println("Failed to create sessionFactory object." + ex);

            throw new ExceptionInInitializerError(ex);

        }


        ManageEmployee ME = new ManageEmployee();


        /* Add few employee records in database */

        Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);

        Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);

        Integer empID3 = ME.addEmployee("John", "Paul", 10000);


        /* List down all the employees */

        ME.listEmployees();


        /* Update employee's records */

        ME.updateEmployee(empID1, 5000);
```

```
   /* Delete an employee from the database */

   ME.deleteEmployee(empID2);


   /* List down new list of the employees */

   ME.listEmployees();

}

/* Method to CREATE an employee in the database */

public Integer addEmployee(String fname, String lname, int salary){

   Session session = factory.openSession( new MyInterceptor() );

   Transaction tx = null;

   Integer employeeID = null;

   try{

      tx = session.beginTransaction();

      Employee employee = new Employee(fname, lname, salary);

      employeeID = (Integer) session.save(employee);

      tx.commit();

   }catch (HibernateException e) {

      if (tx!=null) tx.rollback();

      e.printStackTrace();

   }finally {

      session.close();

   }

   return employeeID;

}

/* Method to  READ all the employees */

public void listEmployees( ){

   Session session = factory.openSession( new MyInterceptor() );

   Transaction tx = null;

   try{

      tx = session.beginTransaction();

      List employees = session.createQuery("FROM Employee").list();

      for (Iterator iterator =

                   employees.iterator(); iterator.hasNext();){

         Employee employee = (Employee) iterator.next();

         System.out.print("First Name: " + employee.getFirstName());
```

```
         System.out.print("  Last Name: " + employee.getLastName());

         System.out.println("  Salary: " + employee.getSalary());

      }
      tx.commit();

   }catch (HibernateException e) {

      if (tx!=null) tx.rollback();

      e.printStackTrace();

   }finally {

      session.close();

   }

}

/* Method to UPDATE salary for an employee */

public void updateEmployee(Integer EmployeeID, int salary ){

   Session session = factory.openSession( new MyInterceptor() );

   Transaction tx = null;

   try{

      tx = session.beginTransaction();

      Employee employee =

               (Employee)session.get(Employee.class, EmployeeID);

      employee.setSalary( salary );

        session.update(employee);

      tx.commit();

   }catch (HibernateException e) {

      if (tx!=null) tx.rollback();

      e.printStackTrace();

   }finally {

      session.close();

   }

}

/* Method to DELETE an employee from the records */

public void deleteEmployee(Integer EmployeeID){

   Session session = factory.openSession( new MyInterceptor() );

   Transaction tx = null;

   try{

      tx = session.beginTransaction();

      Employee employee =
```

```
                (Employee)session.get(Employee.class, EmployeeID);
         session.delete(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
}
```

Compilation and Execution

Here are the steps to compile and run the above mentioned application. Make sure, you have set PATH and CLASSPATH appropriately before proceeding for the compilation and execution.

- Create hibernate.cfg.xml configuration file as explained in configuration chapter.
- Create Employee.hbm.xml mapping file as shown above.
- Create Employee.java source file as shown above and compile it.
- Create MyInterceptor.java source file as shown above and compile it.
- Create ManageEmployee.java source file as shown above and compile it.
- Execute ManageEmployee binary to run the program.

You would get the following result, and records would be created in the EMPLOYEE table.

```
$java ManageEmployee
.......VARIOUS LOG MESSAGES WILL DISPLAY HERE........


Create Operation
preFlush
postFlush
Create Operation
preFlush
postFlush
Create Operation
preFlush
postFlush
```

```
First Name: Zara   Last Name: Ali   Salary: 1000

First Name: Daisy  Last Name: Das   Salary: 5000

First Name: John   Last Name: Paul  Salary: 10000

preFlush

postFlush

preFlush

Update Operation

postFlush

preFlush

postFlush

First Name: Zara   Last Name: Ali   Salary: 5000

First Name: John   Last Name: Paul  Salary: 10000

preFlush

postFlush
```

If you check your EMPLOYEE table, it should have the following records:

```
mysql> select * from EMPLOYEE;

+----+------------+-----------+--------+
| id | first_name | last_name | salary |
+----+------------+-----------+--------+
| 29 | Zara       | Ali       |   5000 |
| 31 | John       | Paul      |  10000 |
+----+------------+-----------+--------+
2 rows in set (0.00 sec

mysql>
```