Hibernate Tutorial

High-performance

Object/Relational persistence

Query service

Licensed under the open source GNU Lesser General Public License (LGPL)

**What is JDBC?** (**Java Database Connectivity**)

Provides a set of Java API for accessing the relational databases from Java program.

## **Pros and Cons of JDBC**

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

## **Why Object Relational Mapping (ORM)?**

As there is a mismatch between the object model and the relational database.

**RDBMSs =>** tabular format

**Object-oriented languages =>** interconnected graph of objects

For example if there is a structure of Employee entity having attributes as

Id, first\_name, last\_name, salary

**The Class representation would be :-**

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;

}

//Getter and Setters

}

**The RDBMSs representation would be:-**

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)

);

But, if we do these both separately, for our application

Means we will have the classes design in our application, and we will jump into database to create table structure.

**We may face several problems here:-**

What if we need to modify the design of our database after few days of development?

Loading and storing objects in a relational database exposes us to the following five mismatch problems –

|  |  |
| --- | --- |
| **Sr.No.** | **Mismatch & Description** |
| 1 | **Granularity**  Sometimes you will have an object model, which has more classes than the number of corresponding tables in the database. |
| 2 | **Inheritance**  RDBMSs do not define anything similar to Inheritance, which is a natural paradigm in object-oriented programming languages. |
| 3 | **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)). |
| 4 | **Associations**  Object-oriented languages represent associations using object references whereas an RDBMS represents an association as a foreign key column. |
| 5 | **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) is a programming technique for converting data between relational databases and object oriented programming languages.

An ORM system has the following advantages over plain JDBC −

|  |  |
| --- | --- |
| **Sr.No.** | **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. |

**Hibernate** is an **O**bject-**R**elational **M**apping (ORM) solution for JAVA.

**Created by :-** Gavin King in 2001

It is an open source persistent framework.

Hibernate maps Java classes to database tables.

And from Java data types to SQL data types.

Hibernates comes in between our **Java POJO classes** and **Database Servers**.

So that it can persists those objects inside database.

## **Hibernate Advantages**

Hibernate uses XML files to map between Java classes and database tables.

So, if there is a change in database structure we just need to modify our XML file.

It reduces SQL work and provides to work around familiar Java Objects.

Provides simple querying of data.

Hibernate supports almost all the major RDBMS in market.

# Hibernate - Architecture

Hibernate has a layered architecture

Following is a very high level view of the Hibernate Application Architecture.



Following is a detailed view :-



Hibernate uses various existing Java APIs, like

Java Database Connectivity (JDBC),

Java Transaction API (JTA), and

Java Naming and Directory Interface (JNDI).

JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

## **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.

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.

It is usually created during application start up

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.

It is to be instantiated each time an interaction is needed with the database.

Persistent objects are saved and retrieved through a Session object.

But they are not usually thread safe, So they should be created and destroyed as needed.

## **Transaction Object**

A Transaction represents a unit of work with the database

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.

**To setup Hibernate :**

Download the hibernate from <http://www.hibernate.org/downloads>.

Unzip the .zip file.

Install the required library files in Classpath.

Hibernate also need some prerequisite jars, which also you can keep

dom4j, Xalan, Xerces, cglib, log4j, Commons, SLF4J

Hibernate has already specified the standard Java properties file called **hibernate.properties**, or as an XML file named **hibernate.cfg.xml**.

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 −

|  |  |
| --- | --- |
| **Sr.No.** | **Properties & 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 autocommit 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 −

|  |  |
| --- | --- |
| **Sr.No.** | **Properties & 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. |

Sample hibernate.cfg.xml file with MySQL database

<?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>

<!—the dialect and driver class must be correct -->

<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, this are the POJO class configuration files -->

<mapping resource = "Employee.hbm.xml"/>

</session-factory>

</hibernate-configuration>

Sessions :

The main function of the Session is to offer, create, read, and delete operations for instances of mapped entity classes.

**States :-**

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.

Few important methods :

|  |  |
| --- | --- |
| **Sr.No.** | **Session Methods & 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. |

**Persistent Class :**

The entire concept of Hibernate is to take the values from Java class attributes and persist them to a database table.

A mapping XML 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. It follows **POJO** implementation of classes.

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.

**Example of 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;

}

// Getters & Setters

}

**The Mapping Files:**

An Object/relational mappings are usually defined in an XML document.

We can write the Mapping files by our own or we can use the tools as well.

Like :-

**XDoclet, Middlegen** and **AndroMDA**

<?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.

Ex: Employee.hbm.xml

**<hibernate-mapping>** as the root element, which contains all the **<class>**elements.

**<class>** elements are used to define specific mappings from a Java classes to the database tables.

**<meta>** element is optional, used for description

**<id>** element maps the unique ID, primary key

**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.

**<generator>** element within the id element is used to generate the primary key values automatically.

**class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence**, **hilo or other** algorithm to create primary key depending upon the capabilities of the underlying database.

All the generator classes implements the **org.hibernate.id.IdentifierGenerator interface**.

 The application programmer may create one's own generator classes by implementing the IdentifierGenerator interface.

Hibernate framework provides many built-in generator classes:

1. **assigned** – default <generator **class**="assigned"></generator>
2. **increment** – generates the unique id only if no other process is inserting data into this table. It generates **short**, **int** or **long** type identifier. If a table contains an identifier then the application considers its maximum value else the application consider that the first generated identifier is 1. For each attribute value, the hibernate increment the identifier by 1. <generator **class**="increment"></generator>
3. **sequence** - It uses the sequence of the database. If there is no sequence defined, it creates a sequence automatically e.g. in case of Oracle database, it creates a sequence named HIBERNATE\_SEQUENCE. In case of Oracle, DB2, SAP DB, Postgre SQL or McKoi, it uses sequence but it uses generator in interbase.

<generator **class**="sequence"></generator>

For defining your own sequence, use the param subelement of generator.

<generator **class**="sequence">

<param name="sequence">your\_sequence\_name</param>

</generator>

1. **hilo** - It uses high and low algorithm to generate the id of type short, int and long.
2. **native** - It uses identity, sequence or hilo depending on the database vendor.
3. **identity** - It is used in Sybase, My SQL, MS SQL Server, DB2 and HypersonicSQL to support the id column. The returned id is of type short, int or long. It is responsibility of database to generate unique identifier.
4. **seqhilo** - It uses high and low algorithm on the specified sequence name. The returned id is of type short, int or long.
5. **uuid** - It uses 128-bit UUID algorithm to generate the id. The returned id is of type String, unique within a network (because IP is used). The UUID is represented in hexadecimal digits, 32 in length.
6. **guid** - It uses GUID generated by database of type string. It works on MS SQL Server and MySQL.
7. **select** - It uses the primary key returned by the database trigger.
8. **foreign** - It uses the id of another associated object, mostly used with <one-to-one> association.
9. **sequence-identity** - It uses a special sequence generation strategy. It is supported in Oracle 10g drivers only.

**<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

**Mapping 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.

## **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 |

**Now the Example :-**

**Create POJO Classes (see above)**

**Create Mapping Configuration File (see above)**

**Create Application Class**

// Application class

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();

}

}

}

Create hibernate.cfg.xml

Execute ManageEmployee binary to run the program.

and records would be created in the EMPLOYEE table.

Other than above simple mapping example,

There 3 most important mapping topics :-

* 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.

**java.util.Set :-**

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. 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.

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; // Set variable

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()))) {

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 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>

Save it as - Employee.hbm.xml.

* 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();

}

}

}

[**java.util.SortedSet**](https://www.tutorialspoint.com/hibernate/hibernate_sortedset_mapping.htm) :-

does not contain any duplicate element and elements are ordered using their natural ordering or by a comparator provided.

mapped with a <set>

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.

private SortedSet certificates;

Certificate class : (Using Comparable and implementing compareTo() )

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);

}

}

}

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>

<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>

* 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.

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;

}

}

}

Application class :

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();

}

}

}

You can try by changing your mapping file, simply set **sort="natural"** and execute your program and compare the results.

[**java.util.List**](https://www.tutorialspoint.com/hibernate/hibernate_list_mapping.htm) :-

stores elements in sequence and allow duplicate elements.

private List certificates;

Certificate class :-

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;

}

}

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>

<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>

* 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.

Application Class :-

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();

}

}

}

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.

[**java.util.Collection**](https://www.tutorialspoint.com/hibernate/hibernate_bag_mapping.htm) :-

also known as the Bag mappings.

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

initialized with java.util.ArrayList.

In Employee.java use -

private Collection certificates;

Create POJO Certificate.java

Now Hibernate 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>

<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>

* 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.

Application class :-

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();

}

}

}

[**java.util.Map**](https://www.tutorialspoint.com/hibernate/hibernate_map_mapping.htm) :-

A **Map** is a java collection that stores elements in key-value pairs and does not allow duplicate elements in the list.

unordered map can be initialized with java.util.HashMap.

at Employee.java :-

private Map certificates;

Create Certificate.java as POJO

Hibernate mapping :-

<?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>

* 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 ie. 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 a Employee parent associated with it. You can use either **<one-to-one>**, **<many-to-one>** or **<many-to-many>** elements based on your requirement.

Application class:

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();

}

}

}

[**java.util.SortedMap**](https://www.tutorialspoint.com/hibernate/hibernate_sortedmap_mapping.htm) :-

A **SortedMap** is a similar java collection as **Map** that stores elements in key-value pairs and provides a total ordering on its keys.

At Employee.java :

private SortedMap certificates;

At Certificate.java :-

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.

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);

}

}

}

Hibernate 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>

<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>

* 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 ie. 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 a 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;

}

}

}

Application class:-

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();

}

}

}

## **Association Mappings**

The mapping of associations between entity classes and the relationships between tables is the soul of ORM.

An association mapping can be unidirectional as well as bidirectional.

[**Many-to-One**](https://www.tutorialspoint.com/hibernate/hibernate_many_to_one_mapping.htm) :-

where an Object can be associated with multiple objects.

Employee.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;

}

}

Address.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;

}

}

Hibernate mapping :-

<?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>

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.

Application class :-

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();

}

}

}

[**One-to-One**](https://www.tutorialspoint.com/hibernate/hibernate_one_to_one_mapping.htm) :-

A **one-to-one** association is similar to **many-to-one** association with a difference that the column will be set as unique.

Keep the Employee.java & Address.java as same as above example

Hibernate mapping :-

<?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>

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.

Application class :-

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();

}

}

}

[**One-to-Many**](https://www.tutorialspoint.com/hibernate/hibernate_one_to_many_mapping.htm) :-

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.

At Employee.java :-

private Set certificates;

At Certificate.java :-

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()))) {

return true;

}

return false;

}

public int hashCode() {

int tmp = 0;

tmp = ( id + name ).hashCode();

return tmp;

}

}

Hibernate mapping :-

<?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>

* 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** 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.

Application class: -

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();

}

}

}

[**Many-to-Many**](https://www.tutorialspoint.com/hibernate/hibernate_many_to_many_mapping.htm) :-

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 manyto-many mapping.

Employee.java :

private Set certificates;

Certificate.java as above program. With equals & hashcode

Hibernate mapping :-

<?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>

* 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 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 ie. 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.

Application class :-

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();

}

}

}

## **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**.

[**Component Mappings**](https://www.tutorialspoint.com/hibernate/hibernate_component_mappings.htm) :-

At employee.java

private Address address;

Address.java as POJO

Hibernate mapping :-

<?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>

The **<component>** element sets the existence of different attributes of Address class inside Employee classes.

Application class:

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();

}

}

}

**Hibernate Annotations :-**

Hibernate annotations are the newest way to define mappings without the use of XML file.

Annotations are introduced after Java1.5

need to install the Hibernate 3.x annotations distribution package,

copy **hibernate-annotations.jar,**

**lib/hibernate-comons-annotations.jar** and

**lib/ejb3-persistence.jar** from the Hibernate Annotations distribution to your CLASSPATH.

Employee.java :-

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.

**@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 allows you to specify the details of the table that will be used to persist the entity in the database.

@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**  Default is key generation strategy.

@Column annotation is used to specify the details of the column to which a field or property will be mapped.

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.

Application class :-

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();

}

}

}

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.

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 –

String hql = "FROM Employee";

Query query = session.createQuery(hql);

List results = query.list();

With fully qualify a class name in HQL -

String hql = "FROM com.hibernatebook.criteria.Employee";

Query query = session.createQuery(hql);

List results = query.list();

As clause –

used to assign aliases

The **AS** keyword is optional and you can also specify the alias directly

String hql = "FROM Employee AS E";

Query query = session.createQuery(hql);

List results = query.list();

String hql = "FROM Employee E";

Query query = session.createQuery(hql);

List results = query.list();

Select Clause –

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 –

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).

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

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.

String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " +

"GROUP BY E.firstName";

Query query = session.createQuery(hql);

List results = query.list();

Named Parameters –

This makes writing HQL queries that accept input from the user easy and you do not have to defend against SQL injection attacks.

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 –

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 –

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.

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.

**avg(property name) :** The average of a property's value

**count(property name or \*) :** The number of times a property occurs in the results

**max(property name) :** The maximum value of the property values

**min(property name) :** The minimum value of the property values

**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 :-**

**Query setFirstResult(int startPosition) :** This method takes an integer that represents the first row in your result set, starting with row 0.

**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();

**Criteria Queries :-**

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.

simplest example of a criteria query

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.

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 conditions

LogicalExpression orExp = Restrictions.or(salary, name);

cr.add( orExp );

// To get records matching with AND conditions

LogicalExpression andExp = Restrictions.and(salary, name);

cr.add( andExp );

List results = cr.list();

**Pagination using Criteria**

**public Criteria setFirstResult(int firstResult)**

**public Criteria setMaxResults(int maxResults)**

Criteria cr = session.createCriteria(Employee.class);

cr.setFirstResult(1);

cr.setMaxResults(10);

List results = cr.list();

**Sorting the Result :-**

The Criteria API provides the **org.hibernate.criterion.Order** class to sort your result set in either ascending or descending order,

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

cr.addOrder(Order.desc("salary"));

// To sort records in ascending order

cr.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.

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"));

**Nativ 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.

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 :-**

String sql = "SELECT \* FROM EMPLOYEE WHERE id = :employee\_id";

SQLQuery query = session.createSQLQuery(sql);

query.addEntity(Employee.class);

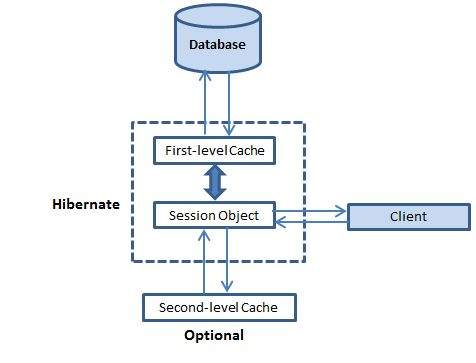
query.setParameter("employee\_id", 10);

List results = query.list();

**Caching :-**

Caching is a mechanism to enhance the performance of a system. It is a buffer memorythat 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.

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.

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.

To use the query cache, you must first activate it using the **hibernate.cache.use\_query\_cache="true"** property in the configuration file.

By 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.

**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.

**EHCache :** It can cache in memory or on disk and clustered caching and it supports the optional Hibernate query result cache.

**OSCache :** Supports caching to memory and disk in a single JVM with a rich set of expiration policies and query cache support.

**warmCache :** A cluster cache based on JGroups. It uses clustered invalidation, but doesn't support the Hibernate query cache.

**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 |

In mapping, we are going to choose EHCache ;-

<?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.

Batch Processing:-

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();

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 POJO Employee class.

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 ;

}

}

Execute ManageEmployee binary to run the program, which will create 100000 records in EMPLOYEE table.

Interceptors :-

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 −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & 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.

## **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 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.

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 {

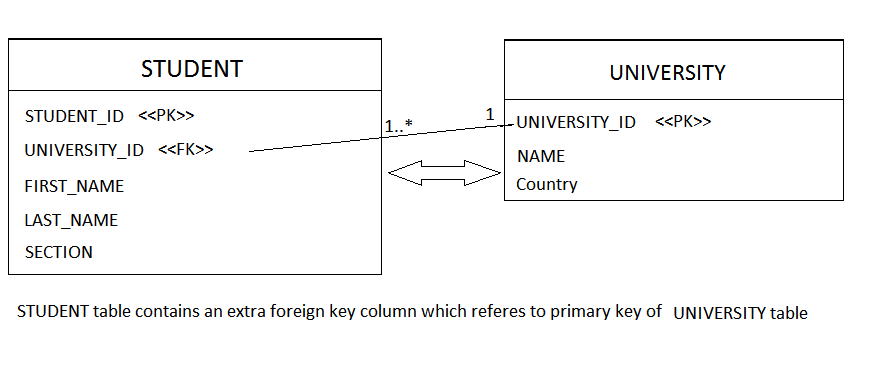
session.close();

}

}

}

# Hibernate Many-To-One Bidirectional (Annotation)

Schema layout for Many-To-One Bidirectional mapping is exactly same as Many-To-One Unidirectional Mapping. One table has a foreign key column that references the primary key of associated table.In Bidirectional relationship, both side navigation is possible.  


We are discussing an example of Student and University relationship.

Many student can enroll at one University.

And one University can have many students. Let’s get going.

#### Step 1: Create required Database Table

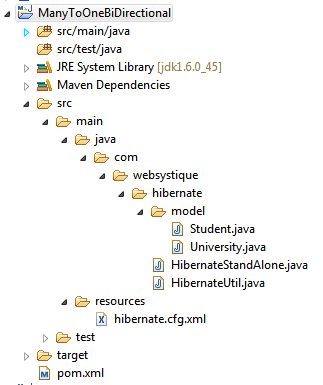
Open MySQL terminal / workbench terminal and execute following MySQL script :

|  |
| --- |
| create table UNIVERSITY (     university\_id BIGINT NOT NULL AUTO\_INCREMENT,     name VARCHAR(30) NOT NULL,     country  VARCHAR(30) NOT NULL,     PRIMARY KEY (university\_id)  );    create table STUDENT (     student\_id BIGINT NOT NULL AUTO\_INCREMENT,     university\_id BIGINT NOT NULL,     first\_name VARCHAR(30) NOT NULL,     last\_name  VARCHAR(30) NOT NULL,     section    VARCHAR(30) NOT NULL,     PRIMARY KEY (student\_id),     CONSTRAINT student\_university FOREIGN KEY (university\_id) REFERENCES UNIVERSITY (university\_id) ON UPDATE CASCADE ON DELETE CASCADE  ); |

Here we have first created University table followed by student table as student table contains a foreign key referring to University table.

#### Step 2: Create project directory structure

Following will be the final project structure:



#### Step 3: Update pom.xml to include required Hibernate and MySQL dependency

Following is the updated minimalistic pom.xml

|  |
| --- |
| <project xmlns="<a class="vglnk" href="<http://maven.apache.org/POM/4.0.0>" rel="nofollow"><span>http</span><span>://</span><span>maven</span><span>.</span><span>apache</span><span>.</span><span>org</span><span>/</span><span>POM</span><span>/</span><span>4</span><span>.</span><span>0</span><span>.</span><span>0</span></a>" xmlns:xsi="<a class="vglnk" href="<http://www.w3.org/2001/XMLSchema-instance>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>w3</span><span>.</span><span>org</span><span>/</span><span>2001</span><span>/</span><span>XMLSchema</span><span>-</span><span>instance</span></a>"      xsi:schemaLocation="<a class="vglnk" href="<http://maven.apache.org/POM/4.0.0>" rel="nofollow"><span>http</span><span>://</span><span>maven</span><span>.</span><span>apache</span><span>.</span><span>org</span><span>/</span><span>POM</span><span>/</span><span>4</span><span>.</span><span>0</span><span>.</span><span>0</span></a> <a class="vglnk" href="<http://maven.apache.org/xsd/maven-4.0.0.xsd>" rel="nofollow"><span>http</span><span>://</span><span>maven</span><span>.</span><span>apache</span><span>.</span><span>org</span><span>/</span><span>xsd</span><span>/</span><span>maven</span><span>-</span><span>4</span><span>.</span><span>0</span><span>.</span><span>0</span><span>.</span><span>xsd</span></a>">      <modelVersion>4.0.0</modelVersion>        <groupId>com.websystique.hibernate</groupId>      <artifactId>ManyToOneBiDirectional</artifactId>      <version>1.0.0</version>      <packaging>jar</packaging>        <name>ManyToOneBiDirectional</name>        <properties>          <hibernate.version>4.3.6.Final</hibernate.version>          <mysql.connector.version>5.1.31</mysql.connector.version>      </properties>        <dependencies>          <!-- Hibernate -->          <dependency>              <groupId>org.hibernate</groupId>              <artifactId>hibernate-core</artifactId>              <version>${hibernate.version}</version>          </dependency>            <!-- MySQL -->          <dependency>              <groupId>mysql</groupId>              <artifactId>mysql-connector-java</artifactId>              <version>${mysql.connector.version}</version>          </dependency>      </dependencies>      <build>          <pluginManagement>              <plugins>                  <plugin>                      <groupId>org.apache.maven.plugins</groupId>                      <artifactId>maven-compiler-plugin</artifactId>                      <version>3.2</version>                      <configuration>                          <source>1.6</source>                          <target>1.6</target>                      </configuration>                  </plugin>              </plugins>          </pluginManagement>      </build>    </project> |

On saving pom.xml with above content, Eclipse will download all the dependencies.

#### Step 4: Create Model classes

Model class **Student** & **University** are simple POJO class which is annotated with JPA annotations to map it to a database tables(created in step 1).

package com.websystique.hibernate.model;

import java.util.List;

import javax.persistence.CascadeType;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.OneToMany;

import javax.persistence.Table;

@Entity

@Table(name = "UNIVERSITY")

public class University {

    @Id

    @GeneratedValue

    @Column(name = "UNIVERSITY\_ID")

    private long id;

    @Column(name = "NAME")

    private String name;

    @Column(name = "COUNTRY")

    private String country;

    @OneToMany(mappedBy = "university", cascade = CascadeType.ALL)

    private List<Student> students;

    public University() {

    }

    public University(String name, String country) {

        this.name = name;

        this.country = country;

    }

    public long getId() {

        return id;

    }

    public void setId(long id) {

        this.id = id;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public String getCountry() {

        return country;

    }

    public void setCountry(String country) {

        this.country = country;

    }

    public List<Student> getStudents() {

        return students;

    }

    public void setStudents(List<Student> students) {

        this.students = students;

    }

    @Override

    public String toString() {

        return "University [id=" + id + ", name=" + name + ", country="

                + country + "]";

    }

}

package com.websystique.hibernate.model;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.ManyToOne;

import javax.persistence.Table;

@Entity

@Table(name = "STUDENT")

public class Student {

    @Id

    @GeneratedValue

    @Column(name = "STUDENT\_ID")

    private long id;

    @Column(name = "FIRST\_NAME")

    private String firstName;

    @Column(name = "LAST\_NAME")

    private String lastName;

    @Column(name = "SECTION")

    private String section;

    @ManyToOne(optional = false)

    @JoinColumn(name = "UNIVERSITY\_ID")

    private University university;

    public Student() {

    }

    public Student(String firstName, String lastName, String section) {

        this.firstName = firstName;

        this.lastName = lastName;

        this.section = section;

    }

    public long getId() {

        return id;

    }

    public void setId(long id) {

        this.id = id;

    }

    public String getFirstName() {

        return firstName;

    }

    public void setFirstName(String firstName) {

        this.firstName = firstName;

    }

    public String getLastName() {

        return lastName;

    }

    public void setLastName(String lastName) {

        this.lastName = lastName;

    }

    public String getSection() {

        return section;

    }

    public void setSection(String section) {

        this.section = section;

    }

    public University getUniversity() {

        return university;

    }

    public void setUniversity(University university) {

        this.university = university;

    }

    @Override

    public int hashCode() {

        final int prime = 31;

        int result = 1;

        result = prime \* result + (int) (id ^ (id >>> 32));

        return result;

    }

    @Override

    public boolean equals(Object obj) {

        if (this == obj)

            return true;

        if (obj == null)

            return false;

        if (!(obj instanceof Student))

            return false;

        Student other = (Student) obj;

        if (id != other.id)

            return false;

        return true;

    }

    @Override

    public String toString() {

        return "Student [id=" + id + ", firstName=" + firstName + ", lastName="

                + lastName + ", section=" + section + "]";

    }

}

Interesting things going on here: Look at following code in University class

|  |
| --- |
| @OneToMany(mappedBy = "university", cascade = CascadeType.ALL)  private List<Student> students; |

@OneToMany on list property here denotes that one University can have multiple students.With students property defined in University class, we can now navigate from University to students. mappedBy says that it’s the inverse side of relationship.Also note the cascade attribute, which means the dependent object(Student) will be persisted/updated/deleted automatically on subsequent persist/update/delete on University object.No need to perform operation separately on Student.

On the other hand, we have following in Student

|  |
| --- |
| @ManyToOne(optional = false)  @JoinColumn(name = "UNIVERSITY\_ID")  private University university; |

@JoinColumn says that Student table will contain a separate column UNIVERSITY\_ID which will eventually act as a foreign key reference to primary key of University table. @ManyToOne says that multiple Student tuples can refer to same University Tuples(Multiple students can register in same university).Additionally , with optional=false we make sure that no Student tuple can exist without a University tuple.

#### Step 5: Create Hibernate configuration file

We need to inform hibernate about how to connect to database, which database dialect we will be using so that hibernate can generate the instruction specific to that database.

We define all these information in hibernate.cfg.xml. Create this file with below content and save it in src/main/resources folder.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <!DOCTYPE hibernate-configuration SYSTEM "<a class="vglnk" href="<http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>hibernate</span><span>.</span><span>org</span><span>/</span><span>dtd</span><span>/</span><span>hibernate</span><span>-</span><span>configuration</span><span>-</span><span>3</span><span>.</span><span>0</span><span>.</span><span>dtd</span></a>">      <hibernate-configuration>      <session-factory>          <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>          <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>          <property name="hibernate.connection.username">myuser</property>          <property name="hibernate.connection.password">mypassword</property>          <property name="hibernate.connection.url">jdbc:<mysql://localhost:3306/websystique></property>          <property name="show\_sql">true</property>          <property name="format\_sql">false</property>          <mapping class="com.websystique.hibernate.model.Student"/>          <mapping class="com.websystique.hibernate.model.University"/>      </session-factory>  </hibernate-configuration> |

#### Step 6: Create Hibernate Utility class

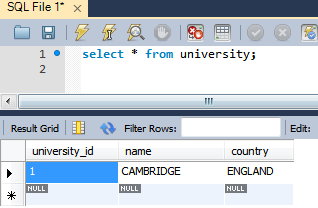
This class is well-known in hibernate community, and used for configuring hibernate on startup and managing session factory.

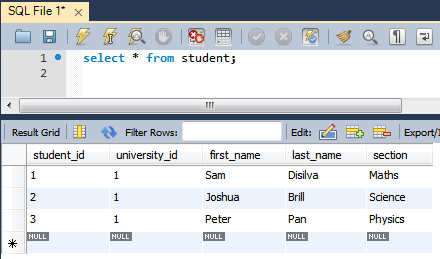
|  |
| --- |
| package com.websystique.hibernate;    import org.hibernate.SessionFactory;  import org.hibernate.cfg.AnnotationConfiguration;    public class HibernateUtil {        private static final SessionFactory sessionFactory;        static{          try{              sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();            }catch (Throwable ex) {              System.err.println("Session Factory could not be created." + ex);              throw new ExceptionInInitializerError(ex);          }      }        public static SessionFactory getSessionFactory() {          return sessionFactory;      }    } |

#### Step 7: Create executable class to Run and perform operations on Database

|  |
| --- |
| package com.websystique.hibernate;    import java.util.ArrayList;  import java.util.List;    import org.hibernate.Session;    import com.websystique.hibernate.model.Student;  import com.websystique.hibernate.model.University;    public class HibernateStandAlone {        @SuppressWarnings("unchecked")      public static void main(String[] args) {            Student student1 = new Student("Sam", "Disilva", "Maths");          Student student2 = new Student("Joshua", "Brill", "Science");          Student student3 = new Student("Peter", "Pan", "Physics");            University university = new University("CAMBRIDGE", "ENGLAND");          List<Student> allStudents = new ArrayList<Student>();            student1.setUniversity(university);          student2.setUniversity(university);          student3.setUniversity(university);            allStudents.add(student1);          allStudents.add(student2);          allStudents.add(student3);            university.setStudents(allStudents);            Session session = HibernateUtil.getSessionFactory().openSession();          session.beginTransaction();            session.persist(university);// Students will be presisted automatically, thanks to CASCADE.ALL defined on students                                      // property of University class.            List<Student> students = (List<Student>) session.createQuery(                  "from Student ").list();          for (Student s : students) {              System.out.println("Student Details : " + s);              System.out.println("Student University Details: "                      + s.getUniversity());          }            // Note that now you can also access the relationship from University to Student            session.getTransaction().commit();          session.close();      }    } |

Below is the snapshot of MySQL database after execution of above program.





What is meant by inverse true in hibernate with example?

**inverse**="**true**" basically **means** that the **inverse** relationship is also mapped within the class **definition** of the other class. ... Hence, **inverse**="**true**" in a **Hibernate mapping** shows that this class (the one with this XML **definition**) is the relationship owner; while the other class is the child.

What is Cascadetype in hibernate?

**CascadeType**.PERSIST : It means that the save() and persist() operations in the**hibernate cascade** to the related entities. **CascadeType**.MERGE : It means that the related entities are joined when the owning entity is joined. ...**CascadeType**.REFRESH : It works similar to the refresh() operation in the**hibernate**.

# Hibernate Inheritance Mapping Tutorial

We can map the inheritance hierarchy classes with the table of the database.

There are three inheritance mapping strategies defined in the hibernate:

1. Table Per Hierarchy
2. Table Per Concrete class
3. Table Per Subclass

#### **Table Per Hierarchy**

In table per hierarchy mapping, single table is required to map the whole hierarchy, an extra column (known as discriminator column) is added to identify the class. But nullable values are stored in the table .

Using XML :-

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3. "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">
6. <hibernate-mapping>
7. <**class** name="com.javatpoint.mypackage.Employee" table="emp121" discriminator-value="emp">
8. <id name="id">
9. <generator **class**="increment"></generator>
10. </id>
12. <discriminator column="type" type="string"></discriminator>
13. <property name="name"></property>
15. <subclass name="com.javatpoint.mypackage.Regular\_Employee" discriminator-value="reg\_emp">
16. <property name="salary"></property>
17. <property name="bonus"></property>
18. </subclass>
20. <subclass name="com.javatpoint.mypackage.Contract\_Employee" discriminator-value="con\_emp">
21. <property name="pay\_per\_hour"></property>
22. <property name="contract\_duration"></property>
23. </subclass>
25. </**class**>
27. </hibernate-mapping>

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
7. //getters and setters
8. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Regular\_Employee **extends** Employee{
4. **private** **float** salary;
5. **private** **int** bonus;
7. //getters and setters
8. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Contract\_Employee **extends** Employee{
4. **private** **float** pay\_per\_hour;
5. **private** String contract\_duration;
7. //getters and setters
8. }

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
6. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">update</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">jtp</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
15. <mapping resource="employee.hbm.xml"/>
16. </session-factory>
18. </hibernate-configuration>

*File: StoreData.java*

1. **package** com.javatpoint.mypackage;
2. **import** org.hibernate.Session;
3. **import** org.hibernate.SessionFactory;
4. **import** org.hibernate.Transaction;
5. **import** org.hibernate.boot.Metadata;
6. **import** org.hibernate.boot.MetadataSources;
7. **import** org.hibernate.boot.registry.StandardServiceRegistry;
8. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
10. **public** **class** StoreData {
11. **public** **static** **void** main(String[] args) {
13. StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
14. Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();
16. SessionFactory factory=meta.getSessionFactoryBuilder().build();
17. Session session=factory.openSession();
19. Transaction t=session.beginTransaction();
21. Employee e1=**new** Employee();
22. e1.setName("Gaurav Chawla");
24. Regular\_Employee e2=**new** Regular\_Employee();
25. e2.setName("Vivek Kumar");
26. e2.setSalary(50000);
27. e2.setBonus(5);
29. Contract\_Employee e3=**new** Contract\_Employee();
30. e3.setName("Arjun Kumar");
31. e3.setPay\_per\_hour(1000);
32. e3.setContract\_duration("15 hours");
34. session.persist(e1);
35. session.persist(e2);
36. session.persist(e3);
38. t.commit();
39. session.close();
40. System.out.println("success");
41. }
42. }

Using annotation :-

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name = "employee101")
6. @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)
7. @DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)
8. @DiscriminatorValue(value="employee")
10. **public** **class** Employee {
11. @Id
12. @GeneratedValue(strategy=GenerationType.AUTO)
14. @Column(name = "id")
15. **private** **int** id;
17. @Column(name = "name")
18. **private** String name;
20. //setters and getters
21. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.\*;
5. @Entity
6. @DiscriminatorValue("regularemployee")
7. **public** **class** Regular\_Employee **extends** Employee{
9. @Column(name="salary")
10. **private** **float** salary;
12. @Column(name="bonus")
13. **private** **int** bonus;
15. //setters and getters
16. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.Column;
4. **import** javax.persistence.DiscriminatorValue;
5. **import** javax.persistence.Entity;
7. @Entity
8. @DiscriminatorValue("contractemployee")
9. **public** **class** Contract\_Employee **extends** Employee{
11. @Column(name="pay\_per\_hour")
12. **private** **float** pay\_per\_hour;
14. @Column(name="contract\_duration")
15. **private** String contract\_duration;
17. //setters and getters
18. }

|  |
| --- |
| Open the hibernate.cgf.xml file, and add entries of entity classes like this: |

1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4. "http://www.hibernate.org/dtd/hibernate-configuration-5.3.dtd">
5. <hibernate-configuration>
6. <session-factory>
8. <property name="hbm2ddl.auto">update</property>
9. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
10. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
11. <property name="connection.username">system</property>
12. <property name="connection.password">jtp</property>
13. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
15. <mapping **class**="com.javatpoint.mypackage.Employee"/>
16. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>
17. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
19. </session-factory>
20. </hibernate-configuration>

*File: StoreTest.java*

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.Session;
4. **import** org.hibernate.SessionFactory;
5. **import** org.hibernate.Transaction;
6. **import** org.hibernate.boot.Metadata;
7. **import** org.hibernate.boot.MetadataSources;
8. **import** org.hibernate.boot.registry.StandardServiceRegistry;
9. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
11. **public** **class** StoreTest {
13. **public** **static** **void** main(String args[])
14. {
15. StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
16. Metadata meta = **new** MetadataSources(ssr).getMetadataBuilder().build();
18. SessionFactory factory=meta.getSessionFactoryBuilder().build();
19. Session session=factory.openSession();
21. Transaction t=session.beginTransaction();
23. Employee e1=**new** Employee();
24. e1.setName("Gaurav Chawla");
26. Regular\_Employee e2=**new** Regular\_Employee();
27. e2.setName("Vivek Kumar");
28. e2.setSalary(50000);
29. e2.setBonus(5);
31. Contract\_Employee e3=**new** Contract\_Employee();
32. e3.setName("Arjun Kumar");
33. e3.setPay\_per\_hour(1000);
34. e3.setContract\_duration("15 hours");
36. session.persist(e1);
37. session.persist(e2);
38. session.persist(e3);
40. t.commit();
41. session.close();
42. System.out.println("success");
43. }
44. }

#### **Table Per Concrete class**

In case of table per concrete class, tables are created as per class. But duplicate column is added in subclass tables.

In case of Table Per Concrete class, there will be three tables in the database having no relations to each other. There are two ways to map the table with table per concrete class strategy.

* By union-subclass element
* By self creating the table for each class

by union-subclass element –

using XML -

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3. "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">
6. <hibernate-mapping>
7. <**class** name="com.javatpoint.mypackage.Employee" table="emp122">
8. <id name="id">
9. <generator **class**="increment"></generator>
10. </id>
12. <property name="name"></property>
14. <union-subclass name="com.javatpoint.mypackage.Regular\_Employee" table="regemp122">
15. <property name="salary"></property>
16. <property name="bonus"></property>
17. </union-subclass>
19. <union-subclass name="com.javatpoint.mypackage.Contract\_Employee" table="contemp122">
20. <property name="pay\_per\_hour"></property>
21. <property name="contract\_duration"></property>
22. </union-subclass>
24. </**class**>
26. </hibernate-mapping>

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
7. //getters and setters
8. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Regular\_Employee **extends** Employee{
4. **private** **float** salary;
5. **private** **int** bonus;
7. //getters and setters
8. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Contract\_Employee **extends** Employee{
4. **private** **float** pay\_per\_hour;
5. **private** String contract\_duration;
7. //getters and setters
8. }

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
6. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">update</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">jtp</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
15. <mapping resource="employee.hbm.xml"/>
16. </session-factory>
18. </hibernate-configuration>

*File: StoreData.java*

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.Session;
4. **import** org.hibernate.SessionFactory;
5. **import** org.hibernate.Transaction;
6. **import** org.hibernate.boot.Metadata;
7. **import** org.hibernate.boot.MetadataSources;
8. **import** org.hibernate.boot.registry.StandardServiceRegistry;
9. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
11. **public** **class** StoreData {
13. **public** **static** **void** main(String[] args) {
15. StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
16. Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();
18. SessionFactory factory=meta.getSessionFactoryBuilder().build();
19. Session session=factory.openSession();
21. Transaction t=session.beginTransaction();
23. Employee e1=**new** Employee();
24. e1.setName("Gaurav Chawla");
26. Regular\_Employee e2=**new** Regular\_Employee();
27. e2.setName("Vivek Kumar");
28. e2.setSalary(50000);
29. e2.setBonus(5);
31. Contract\_Employee e3=**new** Contract\_Employee();
32. e3.setName("Arjun Kumar");
33. e3.setPay\_per\_hour(1000);
34. e3.setContract\_duration("15 hours");
36. session.persist(e1);
37. session.persist(e2);
38. session.persist(e3);
40. t.commit();
41. session.close();
42. System.out.println("success");
43. }
44. }

The **union-subclass** subelement of class, specifies the subclass. It adds the columns of parent table into this table. In other words, it is working as a union.

#### **Table structure for Employee class**

ID

Name

#### **Table structure for Regular\_Employee class**

ID

Name

Salary

Bonus

#### **Table structure for Contract\_Employee class**

ID

Name

PAY\_PER\_HOUR

CONTRACT DURATION

Using Annotation :-

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name = "employee102")
6. @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)
8. **public** **class** Employee {
9. @Id
10. @GeneratedValue(strategy=GenerationType.AUTO)
12. @Column(name = "id")
13. **private** **int** id;
15. @Column(name = "name")
16. **private** String name;
18. //setters and getters
19. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name="regularemployee102")
6. @AttributeOverrides({
7. @AttributeOverride(name="id", column=@Column(name="id")),
8. @AttributeOverride(name="name", column=@Column(name="name"))
9. })
10. **public** **class** Regular\_Employee **extends** Employee{
12. @Column(name="salary")
13. **private** **float** salary;
15. @Column(name="bonus")
16. **private** **int** bonus;
18. //setters and getters
19. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
3. @Entity
4. @Table(name="contractemployee102")
5. @AttributeOverrides({
6. @AttributeOverride(name="id", column=@Column(name="id")),
7. @AttributeOverride(name="name", column=@Column(name="name"))
8. })
9. **public** **class** Contract\_Employee **extends** Employee{
11. @Column(name="pay\_per\_hour")
12. **private** **float** pay\_per\_hour;
14. @Column(name="contract\_duration")
15. **private** String contract\_duration;
17. **public** **float** getPay\_per\_hour() {
18. **return** pay\_per\_hour;
19. }
20. **public** **void** setPay\_per\_hour(**float** payPerHour) {
21. pay\_per\_hour = payPerHour;
22. }
23. **public** String getContract\_duration() {
24. **return** contract\_duration;
25. }
26. **public** **void** setContract\_duration(String contractDuration) {
27. contract\_duration = contractDuration;
28. }
29. }

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
6. <!-- Generated by MyEclipse Hibernate Tools.                   -->
7. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">update</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">jtp</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
16. <mapping **class**="com.javatpoint.mypackage.Employee"/>
17. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
18. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>
19. </session-factory>
20. </hibernate-configuration>

*File: StoreData.java*

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.Session;
4. **import** org.hibernate.SessionFactory;
5. **import** org.hibernate.Transaction;
6. **import** org.hibernate.boot.Metadata;
7. **import** org.hibernate.boot.MetadataSources;
8. **import** org.hibernate.boot.registry.StandardServiceRegistry;
9. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
11. **public** **class** StoreData {
13. **public** **static** **void** main(String[] args) {
15. StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
16. Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();
18. SessionFactory factory=meta.getSessionFactoryBuilder().build();
19. Session session=factory.openSession();
21. Transaction t=session.beginTransaction();
23. Employee e1=**new** Employee();
24. e1.setName("Gaurav Chawla");
26. Regular\_Employee e2=**new** Regular\_Employee();
27. e2.setName("Vivek Kumar");
28. e2.setSalary(50000);
29. e2.setBonus(5);
31. Contract\_Employee e3=**new** Contract\_Employee();
32. e3.setName("Arjun Kumar");
33. e3.setPay\_per\_hour(1000);
34. e3.setContract\_duration("15 hours");
36. session.persist(e1);
37. session.persist(e2);
38. session.persist(e3);
40. t.commit();
41. session.close();
42. System.out.println("success");
43. }
44. }

#### **Table Per Subclass**

In this strategy, tables are created as per class but related by foreign key. So there are no duplicate columns.

In case of Table Per Subclass, subclass mapped tables are related to parent class mapped table by primary key and foreign key relationship.

The **<joined-subclass>** element of class is used to map the child class with parent using the primary key and foreign key relation.

Using XML :-

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
4. "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
5. "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">

8. <hibernate-mapping>
9. <**class** name="com.javatpoint.mypackage.Employee" table="emp123">
10. <id name="id">
11. <generator **class**="increment"></generator>
12. </id>
14. <property name="name"></property>
16. <joined-subclass name="com.javatpoint.mypackage.Regular\_Employee" table="regemp123">
17. <key column="eid"></key>
18. <property name="salary"></property>
19. <property name="bonus"></property>
20. </joined-subclass>
22. <joined-subclass name="com.javatpoint.mypackage.Contract\_Employee" table="contemp123">
23. <key column="eid"></key>
24. <property name="pay\_per\_hour"></property>
25. <property name="contract\_duration"></property>
26. </joined-subclass>
28. </**class**>
29. </hibernate-mapping>

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
7. //getters and setters
8. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
2. **public** **class** Regular\_Employee **extends** Employee{
3. **private** **float** salary;
4. **private** **int** bonus;
6. //getters and setters
7. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Contract\_Employee **extends** Employee{
4. **private** **float** pay\_per\_hour;
5. **private** String contract\_duration;
7. //getters and setters
8. }

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
6. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">update</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">jtp</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
15. <mapping resource="employee.hbm.xml"/>
16. </session-factory>
18. </hibernate-configuration>

*File: StoreData.java*

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.Session;
4. **import** org.hibernate.SessionFactory;
5. **import** org.hibernate.Transaction;
6. **import** org.hibernate.boot.Metadata;
7. **import** org.hibernate.boot.MetadataSources;
8. **import** org.hibernate.boot.registry.StandardServiceRegistry;
9. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
11. **public** **class** StoreData {
12. **public** **static** **void** main(String[] args) {
14. StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
15. Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();
17. SessionFactory factory=meta.buildSessionFactory();
18. Session session=factory.openSession();
20. Transaction t=session.beginTransaction();
22. Employee e1=**new** Employee();
23. e1.setName("Gaurav Chawla");
25. Regular\_Employee e2=**new** Regular\_Employee();
26. e2.setName("Vivek Kumar");
27. e2.setSalary(50000);
28. e2.setBonus(5);
30. Contract\_Employee e3=**new** Contract\_Employee();
31. e3.setName("Arjun Kumar");
32. e3.setPay\_per\_hour(1000);
33. e3.setContract\_duration("15 hours");
35. session.persist(e1);
36. session.persist(e2);
37. session.persist(e3);
39. t.commit();
40. session.close();
41. System.out.println("success");
43. }
44. }

#### **Table structure for Employee class**

ID

Name

#### **Table structure for Regular\_Employee class**

EID

Salary

Bonus

#### **Table structure for Contract\_Employee class**

EID

PAY\_PER\_HOUR

CONTRACT DURATION

Using Annotation :-

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name = "employee103")
6. @Inheritance(strategy=InheritanceType.JOINED)
8. **public** **class** Employee {
9. @Id
10. @GeneratedValue(strategy=GenerationType.AUTO)
12. @Column(name = "id")
13. **private** **int** id;
15. @Column(name = "name")
16. **private** String name;
18. //setters and getters
19. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.\*;
5. @Entity
6. @Table(name="regularemployee103")
7. @PrimaryKeyJoinColumn(name="ID")
8. **public** **class** Regular\_Employee **extends** Employee{
10. @Column(name="salary")
11. **private** **float** salary;
13. @Column(name="bonus")
14. **private** **int** bonus;
16. //setters and getters
17. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.\*;
5. @Entity
6. @Table(name="contractemployee103")
7. @PrimaryKeyJoinColumn(name="ID")
8. **public** **class** Contract\_Employee **extends** Employee{
10. @Column(name="pay\_per\_hour")
11. **private** **float** pay\_per\_hour;
13. @Column(name="contract\_duration")
14. **private** String contract\_duration;
16. //setters and getters
17. }

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
6. <!-- Generated by MyEclipse Hibernate Tools.                   -->
7. <hibernate-configuration>
9. <session-factory>
10. <property name="hbm2ddl.auto">update</property>
11. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
12. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
13. <property name="connection.username">system</property>
14. <property name="connection.password">jtp</property>
15. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
17. <mapping **class**="com.javatpoint.mypackage.Employee"/>
18. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
19. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>
20. </session-factory>
22. </hibernate-configuration>

*File: StoreData.java*

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.Session;
4. **import** org.hibernate.SessionFactory;
5. **import** org.hibernate.Transaction;
6. **import** org.hibernate.boot.Metadata;
7. **import** org.hibernate.boot.MetadataSources;
8. **import** org.hibernate.boot.registry.StandardServiceRegistry;
9. **import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;
11. **public** **class** StoreData {
13. **public** **static** **void** main(String args[])
14. {
15. StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
16. Metadata meta = **new** MetadataSources(ssr).getMetadataBuilder().build();
18. SessionFactory factory=meta.getSessionFactoryBuilder().build();
19. Session session=factory.openSession();
21. Transaction t=session.beginTransaction();
23. Employee e1=**new** Employee();
24. e1.setName("Gaurav Chawla");
26. Regular\_Employee e2=**new** Regular\_Employee();
27. e2.setName("Vivek Kumar");
28. e2.setSalary(50000);
29. e2.setBonus(5);
31. Contract\_Employee e3=**new** Contract\_Employee();
32. e3.setName("Arjun Kumar");
33. e3.setPay\_per\_hour(1000);
34. e3.setContract\_duration("15 hours");
36. session.persist(e1);
37. session.persist(e2);
38. session.persist(e3);
40. t.commit();
41. session.close();
42. System.out.println("success");
43. }
44. }

# Hibernate Lazy Collection

Lazy collection loads the child objects on demand, it is used to improve performance. Since Hibernate 3.0, lazy collection is enabled by default.

To use lazy collection, you may optionally use lazy="true" attribute in your collection. It is by default true, so you don't need to do this. If you set it to false, all the child objects will be loaded initially which will decrease performance in case of big data.

Let's see the hibernate mapping file where we have used lazy="true" attribute.

1. **<list** name="answers" lazy="true"**>**
2. **<key** column="qid"**></key>**
3. **<index** column="type"**></index>**
4. **<one-to-many** class="com.javatpoint.Answer"**/>**
5. **</list>**

The methods of Transaction interface are as follows:

1. **void begin()** starts a new transaction.
2. **void commit()** ends the unit of work unless we are in FlushMode.NEVER.
3. **void rollback()** forces this transaction to rollback.
4. **void setTimeout(int seconds)** it sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
5. **boolean isAlive()** checks if the transaction is still alive.
6. **void registerSynchronization(Synchronization s)** registers a user synchronization callback for this transaction.
7. **boolean wasCommited()** checks if the transaction is commited successfully.
8. **boolean wasRolledBack()** checks if the transaction is rolledback successfully.