Hibernate

# 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 |

# What is ORM?

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

* Lets business code access objects rather than DB tables.
* Hides details of SQL queries from OO logic.
* Based on JDBC 'under the hood'
* No need to deal with the database implementation.
* Entities based on business concepts rather than database structure.
* Transaction management and automatic key generation.
* Fast development of application.

# Some ORM Frameworks:

* Enterprise JavaBeans Entity Beans
* Java Data Objects
* Castor
* TopLink
* Spring DAO
* Hibernate

# Hibernate overview

Hibernate is an Object-Relational Mapping (ORM) solution for JAVA and it raised as an open source persistent framework.

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

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



# Hibernate Advantages:

* Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.
* Provides simple APIs for storing and retrieving Java objects directly to and from the database.
* If there is change in Database or in any table then the only need to change XML file properties.
* Abstract away the unfamiliar SQL types and provide us to work around familiar Java Objects.
* Hibernate does not require an application server to operate.
* Manipulates Complex associations of objects of your database.
* Minimize database access with smart fetching strategies.
* Provides simple querying of data.

# Hibernate Architecture

Following is a detailed view of the Hibernate Application Architecture with few important core classes.



Hibernate uses various existing Java APIs, like JDBC, Java Transaction API (JTA), and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

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

## Configuration Object:

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

* **Database Connection:** This is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml**.
* **Class Mapping Setup :** This component creates the connection between the Java classes and database tables.

## SessionFactory Object:

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

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

## Session Object:

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

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

## Transaction Object:

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

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

## Query Object:

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

## Criteria Object:

Criteria object are used to create and execute object oriented criteria queries to retrieve objects.

# Configuration

Default configuration file name : **hibernate.cfg.xml**

This file is kept in the root directory of your application's classpath.

## Sample hibernate.cfg.xml

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

<!DOCTYPE hibernate-configuration SYSTEM

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

<hibernate-configuration>

<session-factory>

<property name="hibernate.dialect">

org.hibernate.dialect.MySQLDialect

</property>

<property name="hibernate.connection.driver\_class">

com.mysql.jdbc.Driver

</property>

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

<property name="hibernate.connection.url">

jdbc:mysql://localhost/test

</property>

<property name="hibernate.connection.username">

root

</property>

<property name="hibernate.connection.password">

root123

</property>

<!-- List of XML mapping files -->

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

</session-factory>

</hibernate-configuration>

## Hibernate configuration properties

## Properties and Description

**hibernate.dialect :** This property makes Hibernate generate the appropriate SQL for the chosen database.

**hibernate.connection.driver\_class :** The JDBC driver class.

**hibernate.connection.url :** The JDBC URL to the database instance.

**hibernate.connection.username :** The database username.

**hibernate.connection.password :** The database password.

**hibernate.connection.pool\_size :** Limits the number of connections waiting in the Hibernate database connection pool.

**hibernate.connection.autocommit :** Allows autocommit mode to be used for the JDBC connection.

# Sessions

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

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed. The main function of the Session is to offer create, read and delete operations for instances of mapped entity classes. Instances may exist in one of the following three states at a given point in time:

## **transient**:

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

## persistent:

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

## detached:

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

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

SessionFactory factory= null;

Configuration configuration = new Configuration().configure();

final ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySettings(configuration.getProperties()).build();

sessionfactory = configuration.buildSessionFactory(serviceRegistry);\*/

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:

**Transaction beginTransaction() :** Begin a unit of work and return the associated Transaction object.

**Connection close() :** End the session by releasing the JDBC connection and cleaning up.

**Criteria createCriteria(Class persistentClass) :** Create a new Criteria instance, for the given entity class, or a superclass of an entity class.

**Criteria createCriteria(String entityName) :** Create a new Criteria instance, for the given entity name.

**Query createFilter(Object collection, String queryString) :** Create a new instance of Query for the given collection and filter string.

**Query createQuery(String queryString) :** Create a new instance of Query for the given HQL query string.

**SQLQuery createSQLQuery(String queryString) :** Create a new instance of SQLQuery for the given SQL query string.

**void delete(String entityName, Object object) :** Remove a persistent instance from the datastore.

**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.

**Serializable save(Object object) :** Persist the given transient instance, first assigning a generated identifier.

**void update(Object object) :** Update the persistent instance with the identifier of the given detached instance.

# Mapping Files

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

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

public class Employee {

private int id;

private String firstName;

private String lastName;

private int salary;

public Employee() {}

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

this.firstName = fname;

this.lastName = lname;

this.salary = salary;

}

public int getId() {

return id;

}

public void setId( int id ) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName( String first\_name ) {

this.firstName = first\_name;

}

public String getLastName() {

return lastName;

}

public void setLastName( String last\_name ) {

this.lastName = last\_name;

}

public int getSalary() {

return salary;

}

public void setSalary( int salary ) {

this.salary = salary;

}

}

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

create table EMPLOYEE (

id INT NOT NULL auto\_increment,

first\_name VARCHAR(20) default NULL,

last\_name VARCHAR(20) default NULL,

salary INT default NULL,

PRIMARY KEY (id)

);

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

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

<!DOCTYPE hibernate-mapping PUBLIC

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

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

<hibernate-mapping>

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

<meta attribute="class-description">

This class contains the employee detail.

</meta>

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

<generator class="native"/>

</id>

<property name="firstName" column="first\_name" type="string"/>

<property name="lastName" column="last\_name" type="string"/>

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

</class>

</hibernate-mapping>

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

* The mapping document is an XML document having **<hibernate-mapping>** as the root element which contains all the <class> elements.
* The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.
* The **<meta>** element is optional element and can be used to create the class description.
* The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.
* The **<generator>** element within the id element is used to automatically generate the primary key values. Set the **class** attribute of the generator element is set to **native** to let hibernate pick up either**identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.
* The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

# Sample Application Class

We will use this application to save few Employee's records and then we will apply CRUD operations on those records.

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();

}

To save data in DB we use session.save

/\* 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;

}

To read data from DB we can use session.get or we can use the HQL, shown below.

/\* 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();

}

}

To update data, first get the data in entity and then update the entity and close the session or we can use the session.update as shown below.

/\* 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();

}

}

To delete, do session.get and then session.delete

/\* 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();

}

}

}

# Collections Mappings:

## Set Mappings

A Set is a java collection that does not contain any duplicate element. Objects to be added to a set must implement both the equals() and hashCode() methods so that Java can determine whether any two elements/objects are identical. A Set is mapped with a <set> element in the mapping table and initialized with java.util.HashSet. You can use Set collection in your class when there is no duplicate element required in the collection.

### Eg :

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

create table EMPLOYEE (

id INT NOT NULL auto\_increment,

first\_name VARCHAR(20) default NULL,

last\_name VARCHAR(20) default NULL,

salary INT default NULL,

PRIMARY KEY (id)

);

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

create table CERTIFICATE (

id INT NOT NULL auto\_increment,

certificate\_name VARCHAR(30) default NULL,

employee\_id INT default NULL,

PRIMARY KEY (id)

);

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

**Define POJO Classes:**

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

import java.util.\*;

public class Employee {

private int id;

private String firstName;

private String lastName;

private int salary;

private Set certificates;

public Employee() {}

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

this.firstName = fname;

this.lastName = lname;

this.salary = salary;

}

public int getId() {

return id;

}

public void setId( int id ) {

this.id = id;

}

public String getFirstName() {

return firstName;

}

public void setFirstName( String first\_name ) {

this.firstName = first\_name;

}

public String getLastName() {

return lastName;

}

public void setLastName( String last\_name ) {

this.lastName = last\_name;

}

public int getSalary() {

return salary;

}

public void setSalary( int salary ) {

this.salary = salary;

}

public Set getCertificates() {

return certificates;

}

public void setCertificates( Set certificates ) {

this.certificates = certificates;

}

}

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

public class Certificate {

private int id;

private String name;

public Certificate() {}

public Certificate(String name) {

this.name = name;

}

public int getId() {

return id;

}

public void setId( int id ) {

this.id = id;

}

public String getName() {

return name;

}

public void setName( String name ) {

this.name = name;

}

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>

* 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 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 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 \*/

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();

}

}

## List Mapping

A **List** is a java collection that stores elements in sequence and allow duplicate elements. The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index, and search for elements in the list. A List is mapped with a <list> element in the mapping table and initialized with java.util.ArrayList.

### Eg:

**Pojo classes**

public class Employee {

private int id;

private String firstName;

private String lastName;

private int salary;

private List certificates;

//setters and getter

…..

}

public class Certificate{

private int id;

private String name;

//Setters and getters

………..

}

**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>

<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 ie. 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 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 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();

}

}

# Annotations

Hibernate annotations is the newest way to define mappings without a use of XML file. You can use annotations in addition to or as a replacement of XML mapping metadata. If you going to make your application portable to other EJB 3 compliant ORM applications, you must use annotations to represent the mapping information but still if you want greater flexibility then you should go with XML-based mappings.

**Annotated Class Example:**

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 on 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:

The EJB 3 standard annotations are contained in the **javax.persistence** package, so we import this package as the first step. Second we used the **@Entity** annotation to the Employee class which marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

## @Table Annotation:

The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database. The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table. For now we are using just table name which is EMPLOYEE.

## @Id and @GeneratedValue Annotations:

Each entity bean will have a primary key, which you annotate on the class with the **@Id** annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure. By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the **@GeneratedValue** annotation which takes two parameters **strategy** and **generator** which I'm not going to discuss here, so let us use only default the default key generation strategy. Letting Hibernate determine which generator type to use makes your code portable between different databases.

## @Column Annotation:

The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:

* **name** attribute permits the name of the column to be explicitly specified.
* **length** attribute permits the size of the column used to map a value particularly for a String value.
* **nullable** attribute permits the column to be marked NOT NULL when the schema is generated.
* **unique** attribute permits the column to be marked as containing only unique values.

## @Basic

The simplest type of mapping to a database column. The Basic annotation can be applied to a persistent property or instance variable of any type. The use of the Basic annotation is optional for persistent fields and properties of these types. If the Basic annotation is not specified for such a field or property, the default values of the Basic annotation will apply.

**Example 1:**

@[Basic](http://www.objectdb.com/api/java/jpa/Basic)

protected String name;

**Example 2:**

@[Basic](http://www.objectdb.com/api/java/jpa/Basic)(fetch=LAZY)

protected String getName() { return name; }

[FetchType](http://www.objectdb.com/api/java/jpa/FetchType) [**fetch**](http://www.objectdb.com/api/java/jpa/Basic/fetch)

(Optional) Defines whether the value of the field or property should be lazily loaded or must be eagerly fetched. The EAGER strategy is a requirement on the persistence provider runtime that the value must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime. If not specified, defaults to EAGER.

## @Transient

Specifies that the property or field is not persistent. It is used to annotate a property or field of an entity class, mapped superclass, or embeddable class.

**Example:**

@[Entity](http://www.objectdb.com/api/java/jpa/Entity)

public class Employee {

@[Id](http://www.objectdb.com/api/java/jpa/Id) int id;

@[Transient](http://www.objectdb.com/api/java/jpa/Transient) User currentUser;

...

}

## @Temporal

This annotation must be specified for persistent fields or properties of type java.util.Date andjava.util.Calendar. It may only be specified for fields or properties of these types.

The Temporal annotation may be used in conjunction with the [Basic](http://www.objectdb.com/api/java/jpa/Basic) annotation, the [Id](http://www.objectdb.com/api/java/jpa/Id)annotation, or the [ElementCollection](http://www.objectdb.com/api/java/jpa/ElementCollection) annotation (when the element collection value is of such a temporal type.

**Example:**

  @[Temporal](http://www.objectdb.com/api/java/jpa/Temporal)(DATE)

protected java.util.Date endDate;

**TemporalType :**

[TemporalType](http://www.objectdb.com/api/java/jpa/TemporalType) [DATE](http://www.objectdb.com/api/java/jpa/TemporalType/DATE) , TemporalType TIME, TemporalType TIMESTAMP

## @Lob

## @Embeddable

Defines a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity. Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity.

**Example 1:**

@[Embeddable](http://www.objectdb.com/api/java/jpa/Embeddable) public class EmploymentPeriod {

@[Temporal](http://www.objectdb.com/api/java/jpa/Temporal)(DATE) java.util.Date startDate;

@[Temporal](http://www.objectdb.com/api/java/jpa/Temporal)(DATE) java.util.Date endDate;

...

}

## @Embedded

Specifies a persistent field or property of an entity whose value is an instance of an embeddable class. The embeddable class must be annotated as [Embeddable](http://www.objectdb.com/api/java/jpa/Embeddable).

The AttributeOverride, AttributeOverrides, AssociationOverride, and AssociationOverridesannotations may be used to override mappings declared or defaulted by the embeddable class.

**Example:**

@[Embedded](http://www.objectdb.com/api/type/Embedded)

@[AttributeOverrides](http://www.objectdb.com/api/java/jpa/AttributeOverrides)({

@[AttributeOverride](http://www.objectdb.com/api/java/jpa/AttributeOverride)(name="startDate", column=@[Column](http://www.objectdb.com/api/type/Column)("EMP\_START")),

@[AttributeOverride](http://www.objectdb.com/api/java/jpa/AttributeOverride)(name="endDate", column=@[Column](http://www.objectdb.com/api/type/Column)("EMP\_END"))

})

public EmploymentPeriod getEmploymentPeriod() { ... }

## @AttributeOverride

Used to override the mapping of a Basic (whether explicit or default) property or field or Idproperty or field.

May be applied to an entity that extends a mapped superclass or to an embedded field or property to override a basic mapping or id mapping defined by the mapped superclass or embeddable class (or embeddable class of one of its attributes).

## @AttributeOverrides

Used to override mappings of multiple properties or fields.

## @EmbeddedId

## @ElementCollection

## @JoinTable

## @JoinColumns

## @JoinColumn

## @CollectionId

## @OneToOne

Defines a single-valued association to another entity that has one-to-one multiplicity. It is not normally necessary to specify the associated target entity explicitly since it can usually be inferred from the type of the object being referenced. If the relationship is bidirectional, the non-owning side must use the mappedBy element of the OneToOne annotation to specify the relationship field or property of the owning side.

The OneToOne annotation may be used within an embeddable class to specify a relationship from the embeddable class to an entity class. If the relationship is bidirectional and the entity containing the embeddable class is on the owning side of the relationship, the non-owning side must use the mappedBy element of the OneToOne annotation to specify the relationship field or property of the embeddable class.

**Example 1: One-to-one association that maps a foreign key column**

// On Customer class:

@[OneToOne](http://www.objectdb.com/api/java/jpa/OneToOne)(optional=false)

@[JoinColumn](http://www.objectdb.com/api/java/jpa/JoinColumn)(

name="CUSTREC\_ID", unique=true, nullable=false, updatable=false)

public CustomerRecord getCustomerRecord() { return customerRecord; }

// On CustomerRecord class:

@[OneToOne](http://www.objectdb.com/api/java/jpa/OneToOne)(optional=false, mappedBy="customerRecord")

public Customer getCustomer() { return customer; }

## @JoinColumn

## @namedquery

# Examples using Annotations

## @Embedded & @Embeddable

Entity has meaning on its own, that is it can exist on its own. Value object doesn’t have meaning on its own. For Eg, Employee (id, name, Address), here address is value object and employee is entity.

Eg code:

Employee Entity

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** userId;

**private** String userName;

@Embedded

**private** Address address;

//getters and setters

}

Addess Entity

@Embeddable

**public** **class** Address {

**private** String street;

**private** String city;

**private** String state;

//getters and setters

}

The generated schema will contain one table with userid, username, street, city, state columns. We can use the @column annotation in the embeddable class that is Address class.

## @AttributeOverride & @AttributeOverrides

Suppose an employee has 2 addresses, one is homeAddress and other is officeAddress in that case the @Column annotation will not help us in separating the columns names. So to fix this we use the @AttributeOverride & @AttributeOverrides annotations.

Eg :

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** userId;

**private** String userName;

@Embedded

@AttributeOverrides({

@AttributeOverride (name="street", column=@Column(name="Home\_Street")),

@AttributeOverride (name="city", column=@Column(name="Home\_City")),

@AttributeOverride (name="state", column=@Column(name="Home\_State"))

})

**private** Address homeAddress;

@Embedded

**private** Address officeAddress;

//getters and setters

}

The table generated will have columns userid, username, street, city, state, home\_street, home\_city, home\_state

## @ElementCollection

If we want to save more than 2 addresses, multiple addresses. Like if we want to save all the addresses the Employee lived in. In that case we need collection. In DB we will have a separate table for address and userId will be foreign key in that table. In that table we can have multiple addresses for a user.

Employee Model class:

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** userId;

**private** String userName;

@ElementCollection

**private** Set<Address> listOfAddress= **new** HashSet<Address>();

//getters and setters

}

Address Model class:

@Embeddable

**public** **class** Address {

**private** String street;

**private** String city;

**private** String state;

//getters and setters

}

If we save these model objects by putting data in them. The result will be saved in 2 tables.

Employee table: userid, username

Employee\_ listOfAddress table (entity class name + variable name): Employee\_userid, street, city, state

## @JoinTable & @JoinColumn

In the above example if we want to configure the auto generated table name we use JoinTable and to configure the foreign key name we use JoinColumn.

Eg :

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** userId;

**private** String userName;

@ElementCollection

@JoinTable(name="USER\_ADDRESS",joinColumns=@JoinColumn(name="USER\_ID"))

**private** Set<Address> listOfAddress= **new** HashSet<Address>();

//getters and setters

}

Address class is same

## @CollectionId

If we want to have a primary key like address\_id in the USER\_ADDRESS table in that case we should use a collection with indexes that is ArrayList and collectionId annotation

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** userId;

**private** String userName;

@ElementCollection

@JoinTable(name="USER\_ADDRESS",joinColumns=@JoinColumn(name="USER\_ID"))

@GenericGenerator(name="hilogen", strategy="hilo")

@CollectionId(columns = { @Column(name="ADDRESS\_ID") }, generator = "hilogen", type = @Type(type = "long"))

**private** List<Address> listOfAddress= **new** ArrayList<Address>();

//getters and setters

}

## @namedquery

@Entity

@NamedQuery(name="Employee.byId",query="from employee where userid= ?")

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** userId;

**private** String userName;

//getters and setters

}

Session.getNamedQuery(“Employee.byId”)

# HQL

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries which in turns perform action on database.

Although you can use SQL statements directly with Hibernate using Native SQL but I would recommend to use HQL whenever possible to avoid database portability hassles, and to take advantage of Hibernate's SQL generation and caching strategies.

Keywords like SELECT, FROM and WHERE etc. are not case sensitive but properties like table and column names are case sensitive in HQL.

## FROM Clause

You will use **FROM** clause if you want to load a complete persistent objects into memory. Following is the simple syntax of using FROM clause:

String hql = "FROM Employee";

Query query = session.createQuery(hql);

List results = query.list();

If you need to fully qualify a class name in HQL, just specify the package and class name as follows:

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

Query query = session.createQuery(hql);

List results = query.list();

## AS Clause

The **AS** clause can be used to assign aliases to the classes in your HQL queries, specially when you have long queries. For instance, our previous simple example would be the following:

String hql = "FROM Employee AS E";

Query query = session.createQuery(hql);

List results = query.list();

The **AS** keyword is optional and you can also specify the alias directly after the class name, as follows:

String hql = "FROM Employee E";

Query query = session.createQuery(hql);

List results = query.list();

## SELECT Clause

The **SELECT** clause provides more control over the result set than the from clause. If you want to obtain few properties of objects instead of the complete object, use the SELECT clause. Following is the simple syntax of using SELECT clause to get just first\_name field of the Employee object:

String hql = "SELECT E.firstName FROM Employee E";

Query query = session.createQuery(hql);

List results = query.list();

It is notable here that **Employee.firstName** is a property of Employee object rather than a field of the EMPLOYEE table.

## WHERE Clause

If you want to narrow the specific objects that are returned from storage, you use the WHERE clause. Following is the simple syntax of using WHERE clause:

String hql = "FROM Employee E WHERE E.id = 10";

Query query = session.createQuery(hql);

List results = query.list();

## ORDER BY Clause

To sort your HQL query's results, you will need to use the **ORDER BY** clause. You can order the results by any property on the objects in the result set either ascending (ASC) or descending (DESC). Following is the simple syntax of using ORDER BY clause:

String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";

Query query = session.createQuery(hql);

List results = query.list();

If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas as follows:

String hql = "FROM Employee E WHERE E.id > 10 " +

"ORDER BY E.firstName DESC, E.salary DESC ";

Query query = session.createQuery(hql);

List results = query.list();

## GROUP BY Clause

This clause lets Hibernate pull information from the database and group it based on a value of an attribute and, typically, use the result to include an aggregate value. Following is the simple syntax of using GROUP BY clause:

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

"GROUP BY E.firstName";

Query query = session.createQuery(hql);

List results = query.list();

## Using Named Paramters

Hibernate supports named parameters in its HQL queries. This makes writing HQL queries that accept input from the user easy and you do not have to defend against SQL injection attacks. Following is the simple syntax of using named parameters:

String hql = "FROM Employee E WHERE E.id = :employee\_id";

Query query = session.createQuery(hql);

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

List results = query.list();

## UPDATE Clause

Bulk updates are new to HQL with Hibernate 3, and deletes work differently in Hibernate 3 than they did in Hibernate 2. The Query interface now contains a method called executeUpdate() for executing HQL UPDATE or DELETE statements.

The **UPDATE** clause can be used to update one or more properties of an one or more objects. Following is the simple syntax of using UPDATE clause:

String hql = "UPDATE Employee set salary = :salary " +

"WHERE id = :employee\_id";

Query query = session.createQuery(hql);

query.setParameter("salary", 1000);

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

int result = query.executeUpdate();

System.out.println("Rows affected: " + result);

## DELETE Clause

The **DELETE** clause can be used to delete one or more objects. Following is the simple syntax of using DELETE clause:

String hql = "DELETE FROM Employee " +

"WHERE id = :employee\_id";

Query query = session.createQuery(hql);

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

int result = query.executeUpdate();

System.out.println("Rows affected: " + result);

## INSERT Clause

HQL supports **INSERT INTO** clause only where records can be inserted from one object to another object. Following is the simple syntax of using INSERT INTO clause:

String hql = "INSERT INTO Employee(firstName, lastName, salary)" +

"SELECT firstName, lastName, salary FROM old\_employee";

Query query = session.createQuery(hql);

int result = query.executeUpdate();

System.out.println("Rows affected: " + result);

# Association Mappings

## Many-to-One

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

## One-to-One

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

Eg:

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** userId;

**private** String userName;

@OneToOne

@JoinColumn(name="vehicle\_id")

**private** Vehicle vehicle;

//getters and setters

}

@Entity

**public** **class** Vehicle {

@Id @GeneratedValue

**private** **int** vehicleId;

**private** String name;

}

Main class:

Session.save(employee);

Session.save(vehicle);

Tables generated :

Employee : userid, username, vehicle\_vehicleid (after using the @JoinColumn, its name will be vehicle\_id)

Vehicle : vehicleid, name

## One-to-Many

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

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

In the above employee and vehicle example if we make it one to many mapping that is one employee can have many vehicles, we will have to use collections.

classes:

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**private** **int** userId;

**private** String userName;

@OneToMany

@JoinTable(name="user\_vehicle",joinColumns=@JoinColumn(name="user\_id"), inverseJoinColumns=@JoinColumn(name="vehicle\_id"))

**private** List<Vehicle> vehicles= **new** ArrayList<Vehicle>();

//getters and setters

}

@Entity

**public** **class** Vehicle {

@Id @GeneratedValue

**private** **int** vehicleId;

**private** String name;

}

Tables generated :

Employee : userid, username

Vehicle : vehicleid, name

Employee\_Vehicle: employee\_userid, vehicle\_vehicleid

## Many-to-Many

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

# 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 it's own life cycle and completely depends on the life cycle of the owning entity class, then the referred class hence therefore is called as the Component class.

The mapping of Collection of Components is also possible in a similar way just as the mapping of regular Collections with minor configuration differences.

# Criteria

Hibernate provides alternate ways of manipulating objects and in turn data available in RDBMS tables. One of the methods is Criteria API which allows you to build up a criteria query object programmatically where you can apply filtration rules and logical conditions.

The Hibernate **Session** interface provides **createCriteria()** method which can be used to create a **Criteria** object that returns instances of the persistence object's class when your application executes a criteria query.

Following is the simplest example of a criteria query is one which will simply return every object that corresponds to the Employee class.

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

List results = cr.list();

## Restrictions with Criteria

You can use **add()** method available for **Criteria** object to add restriction for a criteria query. Following is the example to add a restriction to return the records with salary is equal to 2000:

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

cr.add(Restrictions.eq("salary", 2000));

List results = cr.list();

Following are the few more examples covering different scenarios and can be used as per 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"));

You can create AND or OR conditions using LogicalExpression restrictions as follows:

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

Criterion salary = Restrictions.gt("salary", 2000);

Criterion name = Restrictions.ilike("firstNname","zara%");

// To get records matching with OR condistions

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

cr.add( orExp );

## Pagination using Criteria:

There are two methods of the Criteria interface for pagination.

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

**public Criteria setMaxResults(int maxResults) :** This method tells Hibernate to retrieve a fixed number maxResults of objects.

Using above two methods together, we can construct a paging component in our web or Swing application. Following is the example which you can extend to fetch 10 rows at a time:

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

cr.setFirstResult(1);

cr.setMaxResults(10);

List results = cr.list();

## Sorting the Results

The Criteria API provides the **org.hibernate.criterion.Order** class to sort your result set in either ascending or descending order, according to one of your object's properties. This example demonstrates how you would use the Order class to sort the result set:

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

// To get records having salary more than 2000

cr.add(Restrictions.gt("salary", 2000));

// To sort records in descening order

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

// To sort records in ascending order

crit.addOrder(Order.asc("salary"));

List results = cr.list();

# Native SQL

You can use native SQL to express database queries if you want to utilize database-specific features such as query hints or the CONNECT keyword in Oracle. Hibernate 3.x allows you to specify handwritten SQL, including stored procedures, for all create, update, delete, and load operations.

Your application will create a native SQL query from the session with the**createSQLQuery()** method on the Session interface.:

public SQLQuery createSQLQuery(String sqlString) throws HibernateException

After you pass a string containing the SQL query to the createSQLQuery() method, you can associate the SQL result with either an existing Hibernate entity, a join, or a scalar result using addEntity(), addJoin(), and addScalar() methods respectively.

# Interceptor ???

# Caching ???

# Lazy Fetching vs Eager Fatching

The fetch type essentially decides whether or not to load all of the relationships of a particular object/table as soon as the object/table is initially fetched.

An example of this would be as follows, consider this User object:

[view plaincopy to clipboardprint?](https://howtoprogramwithjava.com/hibernate-eager-vs-lazy-fetch-type/)

1. **public** **class** User
2. {
3. **import** javax.persistence.OneToOne;
4. **import** javax.persistence.JoinColumn;
6. **private** String username;
7. **private** String password;
8. **private** Profile userProfile;
10. // omitting code for getters and setters for username, password
12. @OneToOne
13. @JoinColumn(name="user\_profile\_id")
14. **private** Profile getUserProfile()
15. {
16. **return** userProfile;
17. }
19. **private** **void** setUserProfile(Profile userProfile)
20. {
21. **this**.userProfile = userProfile;
22. }
23. }

Can you spot the relationship in this User object?

If you can’t, no worries, there’s an easy way to spot it!

Any time you see a @OneToOne, @OneToMany or @ManyToMany annotations, you’ve got a relationship. What’s important to note is that the fetch type should be specified within those annotations, if you don’t specify one then it defaults to FetchType.LAZY.

## Lazy Fetching

Hibernate uses proxy object to support lazy loading. Basically when you load data from tables, hibernate doesn’t load all the mapped objects. As soon as you reference a child or lookup object via getter methods, if the linked entity is not in the session cache, then the proxy code will go to the database and load the linked object. It uses javassist to effectively and dynamically generate sub-classed implementations of your entity objects.

Lazy loading is a technique in which objects are loaded on demand basis. Since Hibernate 3, lazy loading is by default, enabled so that child objects are not loaded when parent is loaded.

## Eager fetching

## Default Fetch type

From the JPA 2.0 spec, the defaults are like so:

OneToMany: LAZY

ManyToOne: EAGER

ManyToMany: LAZY

OneToOne: EAGER

And in hibernate, all is Lazy

# Hibernate 2 vs Hibernate 3

* The Hibernate3 package structure is rooted at org.hibernate instead of net.sf.hibernate.This renaming was done to allow Hibernate2 and Hibernate3 to run side by side in the same application.
* Certain interfaces that have been deprecated in Hibernate3 were moved to the org.hibernate.classic package.
* HibernateException and all other Hibernate exceptions are changed in Hibernate3 as a unchecked runtime exceptions.
* Compared to Hibernate2 , Hibernate3 provides the ParameterizedType interface to allow better re-useability of user type implementations.
* Hibernate3.x wraps Blob and Clob instances, to allow classes with a property of type Blob or Clob to be detached,serialized, deserialized, and passed to merge(). Hibernate2.x not supported this feature.

# Hibernate 3 Features

Compared to Hibernate 2.1 - the most popular object/relational mapping solution in any language - Hibernate 3.0 offers:

* Much more flexible O/R mapping: support for exotic association and inheritance mappings, and greater flexibility when working with legacy data.
* Hibernate3 filters: a unique feature for working with temporal (historical), regional or permissioned data.
* Unprecendented flexibility for mixing handwritten and generated SQL within a single application or even a single entity: full support for \"derived\" entities and attributes defined in the mapping document, full support for overriding any generated SQL statement with handwritten SQL, support for stored procedures.
* Object/Relational/XML mapping: query XML directly from the database for reporting, replicate data between databases via intermediate XML, externalize entity data as XML when interacting with remote systems.
* Enhanced ease of use: better defaulting, an unchecked exceptions model, simplified natural (and composite) key support, simplified CMT integration.
* Enhanced =Criteria= query API: with full support for projection/aggregation and subselects.
* Runtime performance monitoring: via JMX or local Java API, including a second-level cache browser.
* Brand new AST-based HQL parser: bulk update/delete enhancement, better syntax validation.
* *JBoss EJB 3.0 preview: support for annotation-based O/R mappings, full support for EJB-QL 3.0, support for EJB 3.0 =persist()/merge()= lifecycle, JACC-based security model.*
* Hibernate Tools preview: a full suite of Eclipse plugins for working with Hibernate 3.0, including mapping editor, interactive query prototyping, schema reverse engineering tool.
* Many new extension points: including a new, extensible, event-driven architecture
* Documentation enhancements.
* Brand new test suite, including many useful

# Hibernate 4 features

Hibernate 4 new features:

* Move to gradle for builds
* Redesign SessionFactory building
* Introduction of services
* Improved metamodel (not in 4.0.0.Final yet, we planned this, but due to the tasks are more than we expected, and it would take too long to get 4.0 out, so we decided to move this out of 4.0.0.Final but will be upcoming release soon see this for more details, and this is a design document)
* Initial osgi-fication by package splitting (public, internal, spi)
* Support for multi-tenant databases (see this for more details)
* Migration to i18n logging framework (using jboss logging)
* JDK 1.6 (JDBC4) as baseline

# Interview Question