HIBERNATE

Histroy

Hibernate was started in 2001 by Gavin King as an alternative to using EJB2-style entity beans. Its mission back then was to simply offer better persistence capabilities than offered by EJB2 by simplifying the complexities and allowing for missing features.

Early in 2003, the Hibernate development team began Hibernate2 releases which offered many significant improvements over the first release and would go on to catapult HIbernate as the

"de facto" standard for persistence in Java...

Hibernate3, JPA, etc..

What is Hibernate?

- Hibernate is the ORM tool given to transfer the data between a java (object) application and a database (Relational) in the form of the objects.
- Hibernate is the open source light weight tool given by Gavin King, actually JBoss server is also created by this person only.
- Hibernate is a non-invasive framework, means it won't forces the programmers to extend/implement any class/interface, and in hibernate we have all POJO classes so its light weight.
- Hibernate can runs with in or without server, i mean it will suitable for all types of java applications (stand alone or desktop or any servlets etc....)
- Hibernate is purely for persistence (to store/retrieve data from Database).
- Hibernate goal is to relieve the developer from 95 percent of common data persistence related programming tasks.

    ⇨ **Type--ORM S/W (or) ORM Persistence Tech**

    ⇨ **Vendor--SoftTree(RedHat)**

    ⇨ **OpenSource S/w**

    ⇨ **Creator--Mr.Gavin King**

    ⇨ **Versions – 2.X, 3.X, 4.X**

What is ORM (Object Relational Mapping)?

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.

ORM tools provide a host of services thereby allowing developers to focus on the business logic of the application rather than repetitive CRUD (Create Read Update Delete) logic

ORM s/w's:-

EJB Entity beans----------from sun
Hibernate---------from softtree(RedHat)
JPA---from Apache foundation

JDO---from sun

OJB--from Apache foundation

Toplink--- from Oracle s/w

All the ORM S/w internally generates Under Lying DB s/w specific sql queries Dynamically when programmer performs his persistence operations on object's.

What is persistence in java based enterprise applications?

The process of storing enterprise data into the relational database is known as persistence.

Persistence Stores:-

The place where the application data can be stored and managed permanently is called as persistence store.

Ex:-

Files,Database s/w's

Persistence Operations

insert,update and delete & select operations are perfomed on the persistance stores to manipulate it's data and these operations are called as persistance operations.

Persistence Logic

The logic that resides in our s/w applications to interact with persistance stores and to manipulate data of persistance stores through operations is called as persistance logic.

In java application we can use either jdbc(or) ORM persistance logic to perform persistance operation on DB s/w.

DAO:-

It is the Design the Pattern which seperates DataAccessLogic from Bussiness Logic.

DAO classes can be developed with JDBC API,Hibernate,Ibatis Spring/JDBC etc..

What is a FrameWork?

A framework is an abstraction layer on the top of the existing technologies.

A frame work is a semi finished app it provides some predefined support in the application

Development to developers.

A framework is a special kind of software which comes in the form of set of jar files and it makes an application development in a less time.

All the frameworks are non-installble s/w .It means the s/w will be in the form of a set of jar files.

Q.) Features of Hibernate ?

- Hibernate persists java objects into database (Instead of primitives)
- It provides Database services in Database vendor independent Manner, so that java applications become portable across the multiple databases.
- Hibernate generates efficient queries for java application to communicate with Database.
- It provides fine-grained exception handling mechanism. In hibernate we only have Un-checked
- Exceptions, so no need to write try, catch, or no need to write throws (In hibernate we have
- It supports synchronization between in-memory java objects and relational records
- Hibernate provides implicit connection pooling mechanism
- Hibernate supports Inheritance, Associations, Collections
- Hibernate supports a special query language(HQL) which is Database vendor independent
- Hibernate has capability to generate primary keys automatically while we are storing the records into database
- Hibernate addresses the mismatches between java and database
- Hibernate provides automatic change detection Hibernate often reduces the amount of code needed to be written, so it lrnproves the productivity I
- Database objects (tables, views, procedures, cursors, functions ... etc) name changes will not affect hibernate code .
- Supports over 30 dialects
- Hibernate provides caching mechanism for efficient data retrieval
- Lazy loading concept is also included in hibernate so you can easily load objects on start up time
- Getting pagination in hibernate is quite simple.
- Hibernate Supports automatic versioning of rows

What is the main difference between Entity Beans and Hibernate ?

1)In Entity Bean at a time we can interact with only one data Base. Where as in Hibernate we can able to establishes the connections to more than One Data Base. Only thing we need to write one more configuration file.

2) EJB need container like Weblogic, WebSphare but hibernate don't nned. It can be run on tomcat.

3) Entity Beans does not support OOPS concepts where as Hibernate does.

4) Hibernate supports multi level cacheing, where as Entity Beans doesn't.

5) In Hibernate C3P0 can be used as a connection pool.

6) Hibernate is container independent. EJB not.

Hibernate Architecture

The Hibernate architecture includes many objects persistent object, session factory, sessions, mapping files, configuration files .
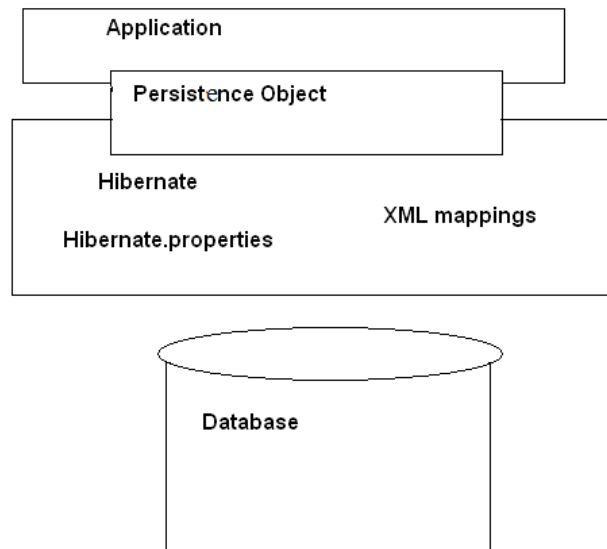
The following diagram shows the high level architecture of Hibernate with mapping file and configuration file.
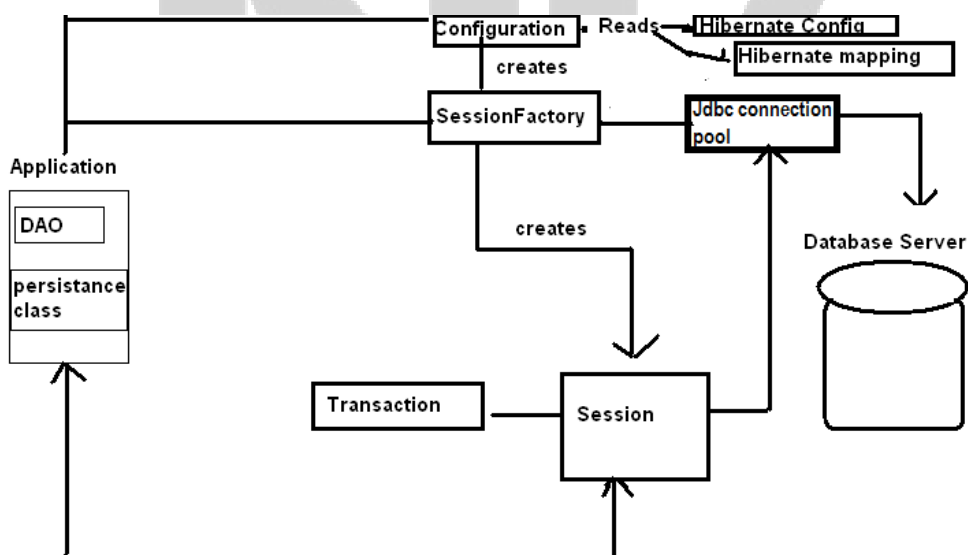
Hibernate framework uses many objects i,e session factory, session, transaction etc. Along with existing Java API such as JDBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface).

What is POJO class? (Plain Old Java Object)

When java class is taken as resource of certain java tech based app and if that class is not extended and not implemented from a predefined class and predefined interfaces of that technology specific API then that java class is called as POJO class.

Ex:-

When java class is taken as the resources of hibernate app and if that is not extending ,implementing predefined classes ,interfaces of hibernate api then that class is called as POJO class.

Required Files in Hibernate Application :

- Any hibernate application must contains the files .

  - ⇨ POJO class/Persistence class/Entity class/Domain class

  - ⇨ Mapping File

  - ⇨ Configuration File

  - ⇨ Client App (One java file to write our logic(main class))

- Actually these are the minimum requirement to run any hibernate application.
- Whether the java application will run in the server or without server, and the application may be desktop or stand alone, swing, awt, servlet. Whatever, but the steps are common to all.
- In order to work with hibernate we don't required any server as mandatory but we need hibernate software (.jar(s) files).
- No Framework is installable Software ,it Means we does n't contain any setup.exe
  When we download any framework,we will get a zip file and we need to unzip it,to get the required jar files,actually all frameworks will follow same common principles like …
  Framework will be in the form of a set of jar files. Every Framework s/w contains two types jar files

  1) Main jar files
  2) Dependent jar files

Each Framework contain one configuration file ,but multiple configuration files also allowed.

We can download hibernate jar files from the following links.Based on requirement we can download the corresponding version

For version 3.x http://sourceforge.net/projects/hibernate/files/hibernate3/

For Version 4.x http://sourceforge.net/projects/hibernate/files/hibernate4/

Note : Along with Hibernate jar's we must include one more jar file,which is nothingbut related to our database,this is depending on your database.For example we are working with Oracle we need to ojdbc6.jar.
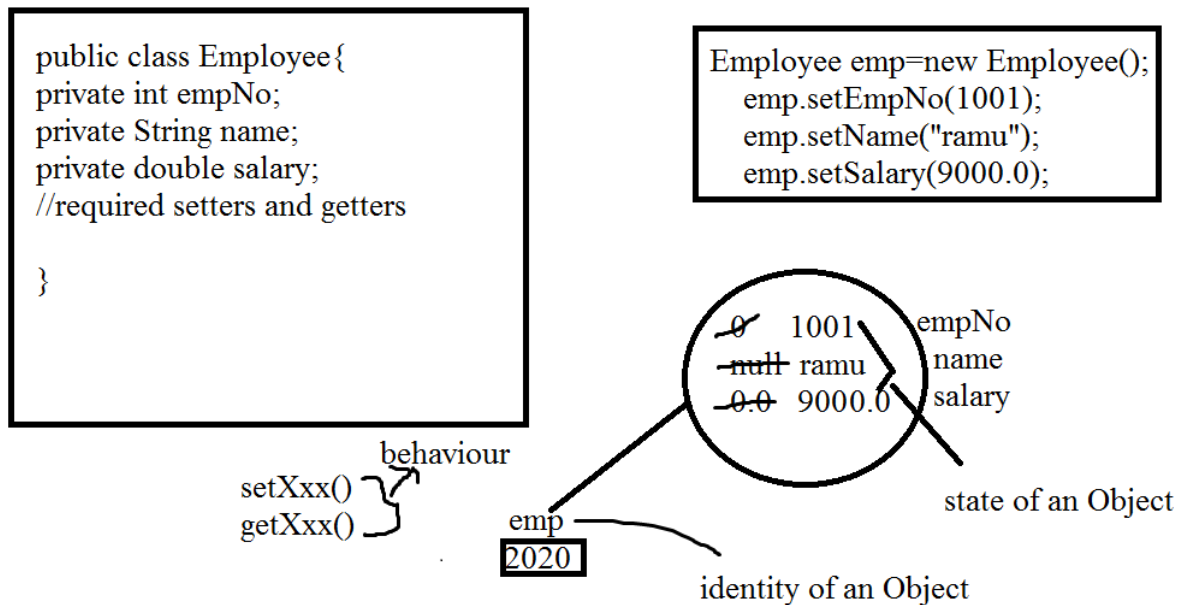
Persistence class/Entity class/pojo class

- It should use java bean mapped with Database table Client App uses this class Object to develop (OR) mapping persistence logic.

- This class Object represents database table record having synchronization b/w them.
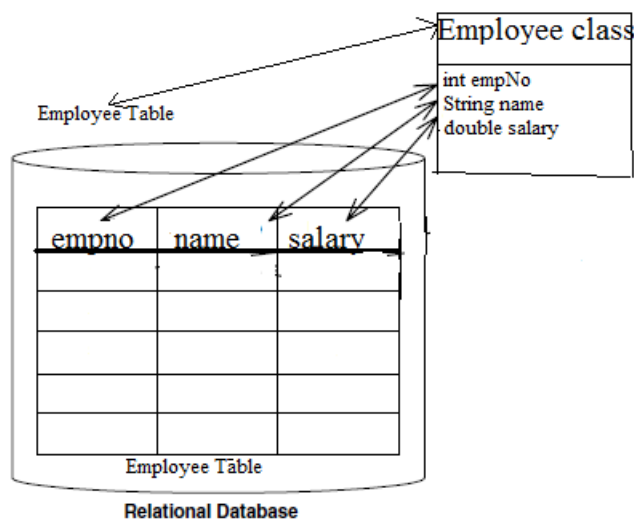
Mapping file :

- In this file hibernate application developer specify the mapping from entity class name to table name and entity properties names to table column names. i.e. mapping of an object oriented data to relational data is done in this file.

- Standard name for this file is<domain-object-name.hbm.xml>

- Mapping can be done using annotations also. If we use annotations for mapping then we no need to write mapping file.

- From hibernate 3.x version on wards it provides support for annotations,So mapping can be done in two ways

  - ❖ XML

  - ❖ Annotations

- Mapping is a meta data ,but not the data. Through mapping meta data hibernate stores (OR) reads the data from database.

- Every ORM tool needs mapping, mapping is the mechanism of placing an object properties(state) into columns of a table.

- Generally an object contains 3 properties like Identity (Object Name) State (Object values) Behavior (Object Methods).

- In ORM, Storing of an Object is nothing but storing the state of an Object, but not the Identity and behavior

For Example:-

```
public class Employee{
private int empNo;
private String name;
private double salary;
//required setters and getters

}
```

```
Employee emp=new Employee();
    emp.setEmpNo(1001);
    emp.setName("ramu");
    emp.setSalary(9000.0);
```

behaviour

setXxx()
getXxx()

emp
2020

0   1001   empNo
null ramu   name
0.0   9000.0 salary

state of an Object

identity of an Object

An ORM tool cannot understand what class Object is needed to stored in what table. To give this information programmer must create/construct mapping.

Mapping between Employee class to Employee Table will be like

Employee Table

Employee class
int empNo
String name
double salary

empno name salary

Employee Table

Relational Database

While constructing a mapping file in hibernate , It is possible to write multiple java classes mapping in a single mapping file.

In Realtime project's , for each domain object we create one mapping file

I,e Number of Entity classes=that many number of mapping files

Syntax Of Mapping xml:

```
<!DOCTYPE ……….  >
<hibernate-mapping>

<class name="Fully qualified name of the class" table="database table Name ">

<id name="variable name" column="column name " type="java/hibernate type" />
```

```
<property name="variable1 name" column="column name" type="java/hibernate type" />

<property name="variable2 name" column="column name" type="java/hibernate type" />

 <class>

</hibernate-mapping>
```

- Each hibernate mapping file must contain one <id> tag. java object identified uniquely by the <id> tag property.

- <id> tag property corresponding column can be primary key(or)non-primary key in the DataBase

Note:-

- In Mapping file class names and Property names are Case-sensitive .

- But Table Names and Column Names are not case sensitive. When the Property name and column name both are same we no need to give Column attribute.

- When the Persistence class name and table name both are same we no need to give table attribute.

- In the Mapping file not required to map all the properties of the entity (pojo class) and all the columns of table.

- As for our requirement we can configure required properties of the entity   with required columns of the table.

Configuration file:

- Configuration is the file loaded into hibernate application when working with hibernate, this configuration file contains 3 types of information.

  ⇨ Connection Properties

  ⇨ Hibernate Properties

  ⇨ Mapping file name(s)

- We must create one configuration file for each database

- we are going to use, suppose if we want to connect with 2 databases, like Oracle, MySql, then we must create 2 configuration files.

No. of databases we are using = That many number of configuration files

- We can write this configuration in 2 ways…
- xml
- By writing Properties file. We don't have annotations ,Actually in hibernate 2.x we defined this configuration file by writing .properties file, but from 3.x xml came into picture.
- So, finally
- Mapping –> xml, annotations
- Configuration –> xml, .properties (old style)

Syntax Of Configuration xml:

<hibernate-configuration>

<session-factory>

 <!-- Related to the connection START -->

<property name="connection.driver_class">Driver Class Name </property>

<property name="connection.url">URL </property>

<property name="connection.username">user </property>

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

<!-- Related to the connection END -->

 <!-- Related to hibernate properties START -->

<property name="show_sql">true/false</property>

<property name="dialect">Database dialect class</property>

<property name="hbm2ddl.auto">create/update/create-drop</property>

<!-- Related to hibernate properties END-->

 <!-- Related to mapping START-->

<mapping resource="hbm file 1 name .xml" / >

<mapping resource="hbm file 2 name .xml" / >

<!-- Related to the mapping END -->

 </session-factory>

</hibernate-configuration>

Dialect in Hibernate:-

- It is a simple java class,which contains mapping b/w java language data type ana database data type.

- This class contains queries format for predefined hibernate methods.

- Hibernate generates queries for the specific database based on the Dialect class. If you want to shift from one data base to another just change the Dialect class name and connection Details in hibernate.cfg.xml file

- All Dialect classes must extend Dialect(Abstract class)

- Hibernate supports almost 30 dialect classes.

- All the Dialect classes are present in org.hibernate.dialect package

- We can write our own Dialect by extending Dialect class.

Dialect class is used to convert HQL queries into Database specific queries.

The following table showing the Supported database dialects in Hibernate

Database	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
Firebird	org.hibernate.dialect.FirebirdDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Informix	org.hibernate.dialect.InformixDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Ingres	org.hibernate.dialect.IngresDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQL5InnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle 8i	org.hibernate.dialect.Oracle8iDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
Progress	org.hibernate.dialect.ProgressDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Sybase ASE 15.5	org.hibernate.dialect.SybaseASE15Dialect
Sybase ASE 15.7	org.hibernate.dialect.SybaseASE157Dialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect

Note :-

By using Hibernate to perform the basic Persistent operations in the client application we need to implement some basic steps.

The basic steps involved in client Application are

Step 1: *Prepare Configuration Object*

Step 2 : *Build SessionFactory object*

Step 3: *Obtain a Session*

Step 4: *Perform the Persistence operations*

Step 5: *close the Session*

Configuration:

Configuration Object

- The org.hibernate.cfg.Configuration is a class and is the basic element of the Hibernate Api.

- An Object Oriented Representation of hibernate configuration file along with mapping file is known as Configuration object.

- By default Hibernate reads configuration file with the name "hibernate.cfg.xml" which is located in classes folder.

- If we want to change the file name(OR) if we want change the location of

  Ex:-

  Configuration cfg =new Configuration();

  cfg.configure("/com/nareshit/xml/hibernate.cfg.xml");

"hibernate.cfg.xml" then we need to pass user given configuration file name(along with path) to "configure()" method of Configuration class.

The Configuration class configure() method described in the below table

configure()	Use the mappings and properties specified in an application resource named hibernate.cfg.xml.
configure(Document document)	Use the mappings and properties specified in the given XML document.

configure(File configFile)	Use the mappings and properties specified in the given application file.
configure(String resource)	Use the mappings and properties specified in the given application resource.
configure(URL url)	Use the mappings and properties specified in the given document.

All the configure() methods described in the above table returns Configuration object.

Configuration object stores the Configuration file data in different variables.Finally all these variables are grouped and created one high level hibernate object called as SessionFactory object.

So configuration object only meant for creating SessionFactory object.

We can also provide the configuration information programmatically,without writing configuration file.(But it will become HardCoding,So not advisable)

Configuration in a programmatic manner

Configuration configuration = new Configuration();

//Adding the Connection details to Configuration object in Programmatically

configuration.setProperty("connection.driver_class", "oracle.jdbc.driver.OracleDriver");

configuration.setProperty("hibernate.connection.url", "jdbc:oracle:thin:@localhost:1521:XE");

configuration.setProperty("hibernate.connection.username", "system");

configuration.setProperty("hibernate.connection.password", "manager");

//Adding the hibernate properties to Configuration object in Programmatically

configuration.setProperty("hibernate.dialect","org.hibernate.dialect.Oracle9Dialect")

//Adding the mapping files to configuration object programmatically

configuration.addResource("Employee.hbm.xml");

configuration.addResource("PersonalDetails.hbm.xml");

Note : *The Configuration object we can create only once for an application, at startup-time, that is while initializing the application.*

After successfully preparing the configuration object by setting all the configuration parameters and mapping documents location,the configuration object is used to create the SessionFactory object.to do this we can use the buildSessionFactory() method of configuration.

After creating the SessionFactory the configuration object can be discarded.

SessionFactory :

- SessionFactory is an interface, present in "org.hibernate" package.

- SessionFactoryImpl is an implemented class of SessionFactory interface

- SessionFactory is a heavy weight object that has to be created only once per application.

- SessionFactory is not a singleton.

- SessionFactory object provides lightweight session object's.

- Session Factory object is the factory for session objects.

- Generally one sessionfactory should be created for one database.

- When you have multiple databases in your application you should create multiple SessionFactory objects.

  For Example :

  If you are using two databases called mysql and oracle in your hibernate application then you need to build 2 SessionFactory object's.

- Sessionfactory is long live multithreaded object.

- SessionFactory is a ThreadSafe object.

- SessionFactory is an ImmutableObject.

- To build the SessionFactory object we can call buildSessionFactory() method from Configuration class

public SessionFactory buildSessionFactory()

Ex:-

Configuration cfg=new Configuration();

cfg.configure("/hibernate.cfg.xml");

SessionFactory sessionFactory=cfg.buildSessionFactory();
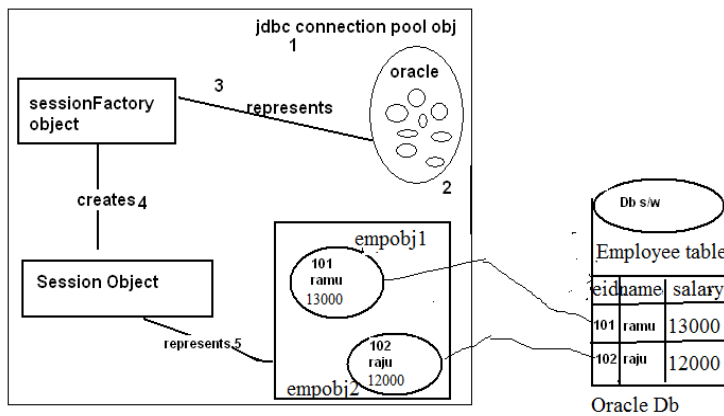
Hibernate software creates Built in Jdbc connection pool having the set of Jdbc connection object's. This connection pool created based on the details given in the Configuration File and will be represented with Hibernate SessionFactory object.

Hibernate Session Object represents Persistence context

Check next chapter for more details about connection pooling

Why Session factory is a heavy weight?

Session factory encapsulates session object,connections,Hibernate properties caching and mappings.

How To Create Session Factory in Hibernate 4

There are many APIs deprecated in the hibernate core framework. One of the frustrating point at this time is, hibernate official documentation is not providing the clear instructions on how to use the new API and it stats that the documentation is in complete. Also each incremental version gets changes on some of the important API.In our earlier versions we have created the session factory as below: -

SessionFactory sessionFactory=cfg.buildSessionFactory();

since Hibernate 4.x, this approach is deprecated. According to Hibernate 4.0 API docs,

 the Configuration class' buildSessionFactory() method is deprecated and it recommends

developers to use the buildSessionFactory(ServiceRegistry) instead.

 Here is the new recommended code that is used to builds the SessionFactory based on a

ServiceRegistry and obtains the Session:

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

ServiceRegistryBuilder registry = new ServiceRegistryBuilder();

registry.applySettings(configuration.getProperties());

ServiceRegistry serviceRegistry = registry.buildServiceRegistry();

SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);

 Session session = sessionFactory.openSession();

In hibernate 4.3 ServiceRegistryBuilder is deprecated so in the new Version's it is recommended to

StandardServiceRegistryBuilder instead of ServiceRegitryBuilder.

Example code :

Configuration configuration = new Configuration().configure("/hibernate.cfg.xml");

ServiceRegistry serviceRegistry   = new StandardServiceRegistryBuilder();

   serviceRegistry .applySettings(configuration.getProperties()).build();

        // builds a session factory from the service registry

SessionFactory   sessionFactory = configuration.buildSessionFactory(serviceRegistry);

Session :-

- Session is an interface present in org.hibernate package and SessionImpl is an implemented class.

- Session object is called persistent manager for the hibernate application.

- It is a single-threaded (not-thread safe), short-lived object

- It encapsulates the functionality of JDBC connection, statement obj

- It has convenience methods to perform persistent operations.

- When a session is opened then internally a connection with the database will be established.

- It is a factory for Transaction objects.

- Session Holds a mandatory (first-level) cache of persistent objects

Note: After we complete the use of the Sesson, It has to be closed, to release all the resources such as associated objects and jdbc connection.

The SessionFactory object is used to open a Hibernate Session .To do this we can use any one of the openSession() methods of SessionFactory.

Methods of SessionFactory interface to open a new Session :

public Session openSession()	This method Prepares a new session that obtains a jdbc connection object by using the connection Manager
public Session openSession(Connection con)	This method prepares a new session that uses the given Connection
public Session openSession(Interceptor icr)	This method prepares a new session that obtains a Jdbc connection object by using the connection manager.The session is registered with the given Interceptor
public Session openSession(Connection con,Interceptor icr)	This method prepares a new session that uses the given Jdbc connection object .The session is

	registered with the given Interceptor

The following code shows for creating a new session :

```
Configuration cfg=new Configuration();

cfg.configure("/hibernate.cfg.xml");

SessionFactory sessionFactory=cfg.buildSessionFactory();

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

The Session interface has various convenience methods to perform the persistence operations.

After obtaining the session we can use the session methods to perform CRUD(Create,Read,Update,Delete) operations on the instances of mapped persisten classes.

```
//get SessionFactory object reference

// After getting the sessionFactory  object we can obtain the session from
sessionFactory

Session session=sessionFactory.openSession();

//Use the session  for performing persistence operations

//After the use of session,close the session

   session.close();
```

After we complete the use of session, it has to be closed to release all the resources such as cached entity object's ,collections,proxies,and a jdbc connection.

But it is recomnded to close the session explicitly by calling   close().

Q. **Develop Hibernate Application,In which we can perform account retrieve operation from Account table ?**

Account Table

accountNumber	name	balance
1001 1002	ramu raju	9000.0 5000.0

```
FirstHibernateApp
     Account.hbm.xml
     Account.java
     hibernate.cfg.xml
     Test.java

     com
         nareshit
             client
                 Test.class

             pojo
                 Account.class
```

Account.java

package com.nareshit.pojo;

public class Account{

        private int accNum;

        private String name;

        private double balance;

        public void setAccNum(int accNum){

this.accNum=accNum;

        }

public void setName(String name){

this.name=name;

}

public void setBalance(double balance){

this.balance=balance;

}

public int getAccNum(){

return accNum;

}

public String getName(){

        return name;

        }

        public double getBalance(){

 return balance;

```
            }

}
```

The Account Persistence class shown in the preceding code declares three persistence fields accNum,name,balance of types int,String,double respectively.

As discussed earlier ,Hibernate requires a small amount of meta data description for persistent class , which is secified using the Account.hbm.xml file .

Account.hbm.xml

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

<!DOCTYPE hibernate-mapping PUBLIC

        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="com.nareshit.pojo.Account"  table="account1">

<id name="accNum">

 <column name="accountNumber"/>

  </id>

 <property name="name">

  <column name="name"  />

     </property>

        <property name="balance" column="balance"/>

  </class>

</hibernate-mapping>
```

The Account.hbm.xml file describes the mappings for Account persistence class. As discussed earlier, It is recommended to save the mapping file name <persistent class name>.hbm.xml.However, it is not mandatory to follow this convention. We can use any filename.

Note :-

- ➢ Xml is a case –sensitive mark-up language.

- ➢ Every Xml file contains the Root tag. And whatever tag we are opened in the xml compulsory we can close.

- ➢ Xml file Contains Either Dtd(Document Type Definition) (OR) XSD(Xml Schema Definition)

- ➢ DTD and XSD are techniques used to frame the rules that are required to construct an xml document

Now ,let us configure the Hibernate configuration file by specifying the configuration parameters such as connection details,hibernate properties and mapping information.

hibernate.cfg.xml

```
<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

      "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE

</property>

<property name="connection.username">system</property>

<property name="connection.password">manager</property>

<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver

</property>

<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

<property  name="show_sql">true</property>

<property name="hbm2ddl.auto">update</property>

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

</session-factory>

</hibernate-configuration>
```

The hibernate.cfg.xml configuration file is using oracle Jdbc thin driver to connect to database for persistence operation. Here we have configured Hibernate properties dialect,show_sql and hbm2ddl.auto properties.

➢ **show-sql**: if the 'show_sql' value is true we can view all the hibernate generated sql queries in the console.

➢ **hbm2ddl.auto**: It has the following values

    a. create   b. update   c. create-drop

a. create: If its value is create while running the application

Case 1: table does not exist

Hibernate creates a new table based on the mapping file configurations

Case 2: table exists

Hibernate Drops the existing table and creates a new table

based on the mapping file configurations

b. Update: If its value is update while running the application

Case 1 : table does n't exist

Create a new table based on the mapping file configurations

Case 2 : table exists

Hibernate Uses the existing table

c.create-drop :

if hbm2ddl.auto property value is create-drop then hibernate creates the a new table when the session-factory object is created.

And drops the table when the sessionFactory is closed .

Test.java

```
package com.nareshit.client;

import com.nareshit.pojo.Account;

import org.hibernate.cfg.Configuration;

import org.hibernate.SessionFactory;

import org.hibernate.Session;

public class Test{

public static void main(String[] args) {

Configuration cfg=new Configuration();

cfg.configure("hibernate.cfg.xml");

SessionFactory factory=cfg.buildSessionFactory();

Session session=factory.openSession();

Account acc=(Account)session.get("con.nareshit.pojo.Account",1001);

if(acc!=null){

System.out.println("Account Number  :"+acc.getAccNum());
```

System.out.println("Holder's Name:"+acc.getName());

System.out.println("Balance : "+acc.getBalance());

}

else{

System.out.println("Account Not Found");

}

session.close();

}

}

To execute the above application we need to set the following jar files in the classpath:

hibernate3.jar

antlr-2.7.6.jar

asm.jar

cglib-2.1.3.jar

commons-collections-2.1.1.jar

commons-logging-1.0.4.jar

ehcache-1.1.jar

dom4j-1.6.1.jar

jta.jar

log4j-1.2.9.jar

Remember: Along with the hibernate jars we must include one more jar file, which is nothing but related to our database, this is depending on your database.

Ojdbc14.jar/classes12.jar file

Session interace methods

load operation/select operation/read operation :-

load/select/read operation is nothing but reading an object from a database into a java application.

In order to perform a load operation/select operation a java programmer has to call any one of the following method from session interface

1. public Object get(String className , Serializable id)
2. public Object get(Class clz , Serializable id)
3. public Object load(String className, Serializable id)

4. public Object load(Class clz, Serializable id)

While loading,Hibernate creates internally an Object of a pojo class for storing the selected data from a database. For creating an Object of pojo class, hibernate internally calls newInstance() of java.lang.Class.

When session.get(className ,id) method is called for loading an Object from database. Hibernate first verifies weather the given id record is exist in the database (OR) not. If exist then immediately reads it from the database and creates persistence Object internally and finally returns the persistence class Object back to java app.

If the given id record is not exist in the database then get(-,-) returns null value back to the java app.

In the case of load(-,-) method, first hibernate creates a pojoclass Object with empty state(data) and returns that Object back to a java program. here the Object returned by hibernate is not a real Object.It means the Object does not contain the data of a database. And it is called as a proxy Object

When we try to access the non-identifier properties from proxy object, at that time Hibernate will hit the database and reads the data from database and put's it into the pojoclass Object this mechanism is called lazy-loading

If the given id does not exist in the database then hibernate throws org.hibernate.ObjectNotFoundException

What is the difference between load() and get() methods?

-->If the given id does not exist in the database then load method throws an Exception but get() method returns null.

-->load() loads the data from database on demand. It is supporting lazy loading.

But get() immediately loads the data from database so it is early loading.

Example to call the get(-,-) method :

```
Session session=sessionFactory.openSession();
Account account=(Account)session.get(Account.class,1001);
//At this line put a break point.... Now observe the console ,After this line executed,
//We can find select statement .
//And the account object initialized with database data
System.out.println(account.getName());
```

Example to call the load (-,-) method:

```
    Session session=sessionFactory.openSession();
     try{
   Account account=(Account)session.load(Account.class,1001);
   //At this line put a break point.... Now observe the console ,After this line executed,//We
   can'not  find select statement .
   //And the account object not  initialized with the database data And the Account is proxy
   object
   System.out.println(account.getName());
 //After this line you can find select query on the console ,And now account object is
 //initialized with database data
 }catch(ObjectNotFoundException e){
 e.printStackTrace();
 }
```

When we use session.get(-,-) method if object is present we have to implement some logic,if not
we need to implement some other logic.

Example :

```
      Session session=sessionFactory.openSession();
      Account account=(Account)session.get(Account.class,1001);
      if(account!=null){
      System.out.println(account.getName());
      System.out.println(account.getBalance());
      }else{
      System.out.println("Account Not Found ");
      }
```

But in the case of load(-,-) method if the given id record is not exist load(-,-) method will rise
org.hibernate.ObjectNotFoundException.So we cannot implement logic like above code .

close() :

public Connection close()throws HibernateException

once the transaction is completed we need to close the session. When we close() the
session all the associated object's with the session will be de-associated from session
and jdbc connection also closed. It is not strictly necessary to close the session but
you must be at least call disconnect() it.

clear():   public void clear()

This method used to de-associate all the object's from session.

evict(Object obj)   : public void evict(Object obj)

This method is used to de-associate the specified object from session

Note :

When a session is opened then internally a connection with the database will be established.

when a session is opened in hibernate then internally a cache (buffer) is opened along with the session.

Hibernate supported two-levels of caching

   1) First-Level Cache (part of session)

   2)   Second-Level Cache(part of SessionFactory)(we will discuss about second level cache in the next chapters)

As a programmer we are not taking any responsibility to create first-level in session.but second-level cahce expilictly configured/created by programmer.

The First level cache is automatically created when a session is opened and automatically closed   when a session is closed.

First level cache associated with session object And is available only till session object is live. It is available to session object only and is **not accessible to any other session object** in any other part of application.

The operations done on the persistence Objects will be stored in a cache,it means if we perform insert (OR) delete (OR) update (OR) select then the result of those operations   will be stored by hibernate internally in the cache of that session.

Hibernate First Level Cache
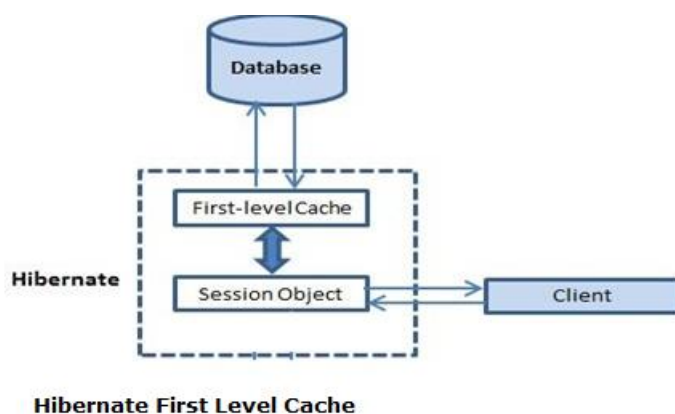
Important Points

1. First level cache is associated with "session" object and other session objects in application can not see it.
2. The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
3. When we read/select/load an object first time, it is retrieved from database and stored in first level cache associated with hibernate session.
4. If we read/select/load same object again with same session object, it will be loaded from cache and no sql query will be executed.
5. The loaded object can be removed from session using session.evict(obj) method. The next time loading of this object will again make a database call if it has been removed using evict(-) method.
6. The whole session cache can be removed using session.clear() method. It will remove all the objects data stored in cache.

For Example :

If we load an Object from a database then first hibernate verifies whether the Object is there in session cache(OR) not . if it is yes ,then hibernate reads the Object from cache instead of Database.so the trips between java app and database will be reduced .

In this example, I am retrieving AccountEntity object from database using hibernate session. I will retrieve it multiple times, and will observe the sql logs to see the differences.

```
//Open the Session
Session session=sessionFactory.openSession();
//fetch the Account entity from database first time
Account acc1=(Account)session.get("com.nareshit.pojo.Account",3001);
System.out.println("Name : "+acc1.getName());
System.out.println("Balance : "+ acc1.getBalance());
System.out.println();
//fetch the  same Account entity  second time
Acc1=(Account)session.get("com.nareshit.pojo.Account",3001);
System.out.println("Name : "+acc1.getName());
System.out.println("Balance : "+acc1.getBalance());

OutPut :
```

```
Hibernate: select account0_.accountId as accountI1_0_0_, accour
Name : ramu
Balance : 8800.0


Name : ramu
Balance : 8800.0
```

you can see that second session.get(-,-) statement does not generated select query again and account object is loaded from first level cache.

Removing cache objects from first level cache example

Though we can not disable the first level cache in hibernate, but we can remove some of objects from it when needed. This is done using session interface methods :

  - evict(object obj)
  - clear()

Here **evict(-) is used to remove a particular object from cache associated with session**, and **clear() method is used to remove all cached objects associated with session**. So they are essentially like remove one and remove all.

```java
    //open the session

  Session session=sessionFactory.openSession();

//fetch the Account entity from database first time

Account
acc1=(Account)session.get("com.nareshit.pojo.Account",3001);
    System.out.println("Name : "+acc1.getName());
    System.out.println("Balance : "+ acc1.getBalance());
    System.out.println();
    //fetch the  same Account entity  second time
  acc1=(Account)session.get("com.nareshit.pojo.Account",3001);
    System.out.println("Name : "+acc1.getName());
    System.out.println("Balance : "+acc1.getBalance());
    System.out.println();
        session.evict(acc1);
      //session.clear();
  //After calling evict(-) method fetch the  same Account
entity  third  time
 acc1=(Account)session.get("com.nareshit.pojo.Account",3001);
    System.out.println("Name : "+acc1.getName());
```

```
    System.out.println("Balance : "+acc1.getBalance());
```

OutPut :

Hibernate: select account0_.accountId as accountI1_0_0_, accour
Name : ramu
Balance : 8800.0

Name : ramu
Balance : 8800.0

Hibernate: select account0_.accountId as accountI1_0_0_, accour
Name : ramu
Balance : 8800.0

you can see that second session.get(-,-) statement does not generated select query again and account object is loaded from first level cache.

And you can see that third session.get(-,-) statement does generated the select query again and account object is loaded from database because with session.evict(acc1); method call is deleted acc1 object from first level cache.

contains(Object obj) :

public boolean contains(Object object)

It is used to check whether the object's is associated with the session OR not.

 Example :

```
Session session=sessionFactory.openSession();
Account account=(Account)session.get("com.nareshit.pojo.Account",3001);
    System.out.println("After calling get() method");
    if(session.contains(account)){
    System.out.println("account is associated with the session ");
    }
    else{
    System.out.println("account is not associated with the session ");
    }
    session.clear();
    System.out.println("After calling clear method() ");
    if(session.contains(account)){
    System.out.println("account is associated with the session ");
    }
    else{
    System.out.println("account is not associated with the session ");
```

```
    }
```

isConnected():

public boolean isConnected();

this method is used to check weather there is a connection is associated with the session (OR) not.

Persistent Object Life Cycle

Persistent Object has three life cycle states

1. Transient state
2. Persistent state
3. Detached state
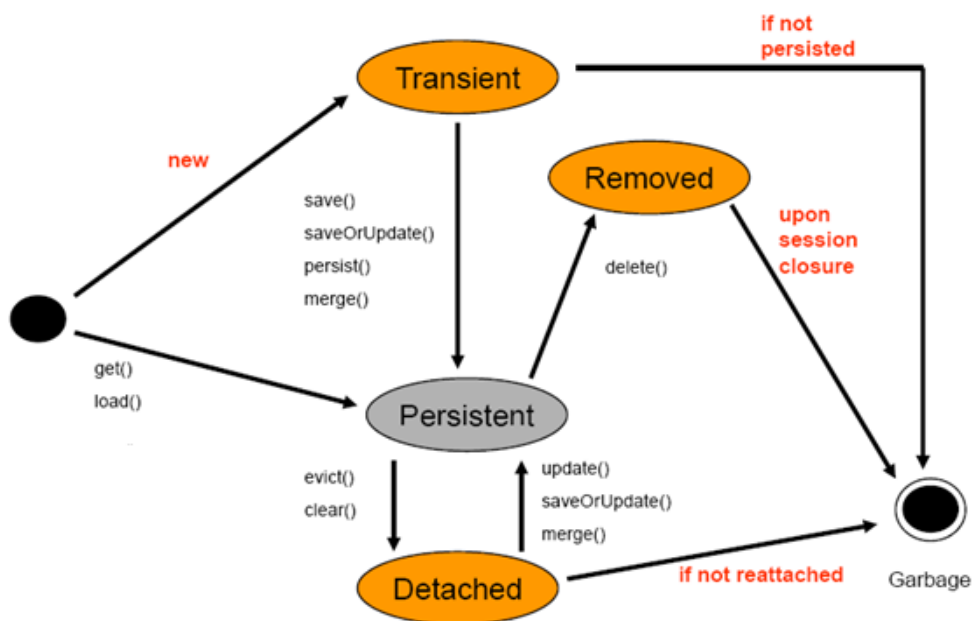
Transient state: An object is said to be in transient state, when it is not associated with session and that object data not present in data base.
Example: Table ACCOUNT

ACCNO	NAME	BALANCE
1001	Pavan	15000.0
1002	Ramu	20000.0

Application code:

Account account = new Account();
account.setAccNum(1003);
account.setName("raju");
account.setBalance(21000);

In the above example account object is not associated with session and there is no matching record in the ACCOUNT table. So we can say that **account object is in transient state.**

In this state object is non-transactional. i.e. object is not synchronized with record. i.e. Modifications done to entity, doesn't save into database.

Persistent state: An object is said to be in persistence state, when it is associated with session as well as object present in database.

Example: Table ACCOUNT

ACCNO	NAME	BALANCE
1001	Pavan	15000.0
1002	Ramu	20000.0

Application Code:

Account account = (Account) session.get(Account.class,1002);

In the above example account object is associated with session and there is a matching record in ACCOUNT table. So we can say that account object is in persistent state.

In this state object is transactional. i.e. the object is synchronized with database record.

Changes made to objects in a persistent state are automatically saved to the database without invoking session persistence methods.

Detached State: An object is said to be in detached state, when the object is not associated with session but present in data base.

Example: Table Account

ACCNO	NAME	BALANCE
1001	Pavan	15000.0
1002	Ramu	20000.0

1003	Raju	210000.0

Application code :

```
Account account = new Account();
 account.setAccNum(1003);
 account.setName("Raju");
account.setBalance(21000);
 //Now the account object is said to be in transient state
 session.save(account);
session.beginTransaction().commit();
 // Now the account object is said to be in persistence state
session.close(); // Now the account object is said to be in detached state.
```

In the above example after calling session.close() method, account object is moved to Detached state from persistent state. As session i s garbage collected, if we try to perform some modifications to entity object those changes will not be stored into database.

Session interface methods

save(-) :-

public Serilalizable save(Object obj)throws HibernateException

save method stores an object into the database. That means it insert an object state into the database . the save() method returns the serializable identifier back to java application.

Example Code :

public void createAccount(){

Session session = sessionFactory.openSession();

Transaction tx=session.beginTransaction();

Account account = new Account();

account.setAccNum(1002); .

account.setName("sathish");

account.setBalance(9000.0);

Integer id=(Integer) session.save (account);

tx.commit();

System.out.println("Account object is created with id :"+id);

//The record with 1002 will be inserted into the database by generating the insert query

}

What is Transaction ?

❖ Transaction used by the application to specify atomic units of work (Transaction management).
❖ Using Session Object we can create Transaction object in two ways.
  • Transaction transaction =session.getTransaction();
  • Transaction transaction = session.beginTransaction();
❖ Transaction object is unique per session object.
❖ Transaction interface defines following methods to deal with transactions.
  • transaction.begin() {Transaction beginning)
  • transaction. commit(); { successful transaction ending)
  • transactin.rollback(); {un successful transaction ending}
❖ Default auto commit value is false in Hibernate.
❖ Default auto commit value is true in JDBC
❖ In hibernate even to execute one DML operation also we need to implement Transactions.
❖ *Hibernate supprots*
  ▪ *JDBC Transaction.*
  ▪ *JTA Transaction.*
  ▪ *Spring Transaction*

Sample Transaction Code is as Follows ...

```
Session  session= sessionFactory.openSession();
Transaction tx = null;
try {
tx = session.beginTransaction();
// DML operations
 tx.commit();}
catch(Exception e) {

tx.rollback();

}

finally{

session.close();

}
}
```

saveOrUpdate(-) method:-

public void saveOrUpdate(Object obj)throws HibernateException

➢ saveOrUpdate(Object obj) method can be used to either INSERT or UPDATE based upon existence of record. Clearly saveOrUpdate(-) is more flexible in terms of use but it involves an extra processing to find out whether record already exists in table or not.
➢ saveOrUpdate() method either INSERT or UPDATE based upon existence of object in database.If persistence object already exists in database then UPDATE SQL will execute and if there is no corresponding object in database than INSERT will run.

 Account Table :

AccNum	Name	Balance

Application Code:-

Example 1

Session session = sessionFactory.openSession();

Transaction tx=session.beginTransaction();

Account account = new Account();

account.setAccNum(1001); .

 account.setName("sathish");

account.setBalance(6800.00);

session.saveOrUpdate(account);

tx.commit();

//Now saveOrUpdate() performs the insertion operation operation

Because record with 1001 id is not available in the database.

After executing the above program table will be updated as follows

Account Table :

AccNum	Name	Balance
1001	sathish	6800.0

Example 2

Session session = sessionFactory.openSession();

Transaction tx=session.beginTransaction();

Account account = new Account();

account.setAccNum(1001);

account.setName("Sathish.Bandi");

account.setBalance(8800.00);

session.saveOrUpdate(account);

session.getTransaction().commit();

//Now saveorupdate() perfoms the update operation

Because record with 1001 id available in the database.

update(Object obj)

public void update(Object obj)throws HibernateException

update method is used for updating the object using identifier. If the identifier is missing or doesn't exist, it will throw exception.

Account Table :

| AccNum	Name	Balance
1001	sathish	5000

Application code:-

Session session = sessionFactory.openSession();

Transaction tx=session.beginTransaction();

Account account = new Account();

account.setAccNum(1001); .

account.setName("sathish.Bandi");

account.setBalance(6800.00);

session.update(account);

tx.commit();

//Now update() performs the update

record with 1001 id available in the database.

After executing the above program table will be updated as follows

AccNum	Name	Balance
1001	Sathish.Bandi	6800.0

What is the Difference between session.update() and session.lock() in Hibernate ?

The session.lock() method simply reattaches the object to the session without checking or updating the database on the assumption that the database in sync with the detached object.
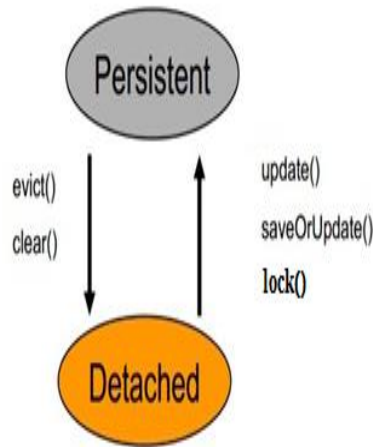
the same persistent instances are reused for each interaction with the database. The application manipulates the state of detached instances originally loaded in another Session and then "reassociates" them using **Session.update()** or **Session.saveOrUpdate()**.

Note: Use session.lock() only if you are absolutely sure that the detached object is in sync with your detached object.

What is the difference between session.update() and session.merge()?

The **session.update()** update the record only when given object related row is available in the table otherwise updae() fails in record updation(if record is not available).

the **session.merge()** tries to update the record if the given pojo object related record is available in the database table and if it is not available this method inserts the record by using given pojo class object data. The given instance does not become associated with the session, it remains detached.

What is the difference between session.saveOrUpdate() and session.merge()?

session.merge(): if there is a persistent instance with the same identifier currently associated with the session, copy the state of the given object onto the persistent instance

if there is no persistent instance currently associated with the session, try to load it from the database, or create a new persistent instance the persistent instance is returned

session.saveOrUpdate(): The session save or update() takes given pojo class object identify field member variable value as criteria value to check wheather value based record is already available in the table or not if available this method perform update operation otherwise this method perform insert operation.

Note: if you are trying to update objects by using merge() then the given instance does not become associated with the session, it remains detached . if you are trying to update objects by using saveOrUPdate() then the given instance become associated with the session.

What is session.flush() method in Hibernate ?

public void flush() throws HibernateException

This method is used to synchronize session data with database.

Example :

Session session = sessionFactory.openSession();

Transaction tx=session.beginTransaction();

Account account=(Account)session.get("com.nareshit.pojo.Account",1001);

account.setName("java sathish ");

account.setBalance(17000.0);

session.flush();

System.out.println("Break….point and observe the console");

System.in.read();

tx.commit();

In the above example when we call session.flush(), Hibernate checks (OR) compares account object data and corresponding record in the database.If it is finds difference ,It will execute update query to update Object data into the database record.

When tx.commit() is called it will also check object data and corresponding record data. If it finds difference it will update Object data into the database.

So after tx.commit() call ,we should not call session.flush() method because when we call tx.commit() method is called session is sync with the database.

What is session.refresh(Object obj) method in Hibernate ?

public void refresh(Object obj) throws HibernateException

This method is used to synchronize the database data with session data.

Wha is the Difference between getCurrentSession() and openSession() in Hibernate ?

getCurrentSession() :

Obtains the current session. The "current session" refers to a Hibernate Session bound by Hibernate behind the scenes, to the transaction scope.

A Session is opened when getCurrentSession() is called for the first time and closed when the transaction ends. It is also flushed automatically before the transaction commits. You can call getCurrentSession() as often and anywhere you want as long as the transaction runs. Only the Session that you obtained with sessionFactory.getCurrentSession() is flushed and closed automatically.

openSession() :
If you decide to use manage the Session yourself the go for sessionFactory.openSession() , you have to flush() and close() it.

It does not flush and close() automatically.
Example :

```
Session session =sessionFactory.openSession();
Transaction tx =session.berginTransaction();

try {
tx.begin();

// Do some work
session.save(...);

session.flush(); // Extra work you need to do

tx.commit();
}
catch (HibernateException e) {
tx.rollback();
}
finally {
session.close(); // Extra work you need to do
}
```

- ◢ 📲 SessionInterfaceMethodsDemo
  - ◢ 🐸 src
    - ◢ ⊞ com.nareshit.client
      - ▷ 🗋 Test.java
    - ◢ ⬔ com.nareshit.config
      - ⊠ hibernate.cfg.xml
    - ◢ ⊞ com.nareshit.dao
      - ▷ 🗋 AccountDao.java
      - ▷ 🗋 AccountDaoImpl.java
    - ◢ ⬔ com.nareshit.mapping
      - ⊠ Account.hbm.xml
    - ◢ ⊞ com.nareshit.pojo
      - ▷ 🗋 Account.java
    - ◢ ⏣ com.nareshit.utility
      - ▷ 🗋 HibernateUtility.java

Account.java

```java
package  com.nareshit.pojo;
public class Account {
        private int accNum;
        private String name;
        private double balance;
        public String toString() {
                return "\naccNumber :"+accNum+"\nname :"+name+"\nbalance :"+balance;
        }
        public int getAccNum() {
                return accNum;
        }
        public void setAccNum(int accNum) {
                this.accNum = accNum;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public double getBalance() {
                return balance;
        }
        public void setBalance(double balance) {
                this.balance = balance;
        }
}
```

Acccount.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
        <class name="com.nareshit.pojo.Account" table="AccountDetails">
```

```
            <id name="accNum" length="12">
                    <generator class="increment" />
            </id>
            <property name="name" length="15" />
            <property name="balance" column="bal" length="10" />
    </class>
</hibernate-mapping>
```

hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
        <session-factory>
                <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver
                </property>
                <property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE
                </property>
                <property name="connection.username">system</property>
                <property name="connection.password">manager</property>
                <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
                <property name="show_sql">true</property>
                <property name="hbm2ddl.auto">update</property>
                <mapping resource="com/nareshit/mapping/Account.hbm.xml" />
        </session-factory>
</hibernate-configuration>
```

AccountDao.java

```java
package com.nareshit.dao;
import com.nareshit.pojo.Account;
public interface AccountDao {
        public int createAccount(Account acc);
        public void updateAccountHolderName(int accNum,String newName);
        public void deleteAccount(int accNum);
        public Account searchAccount(int accNum);
}
```

AccountDaoImpl.java

```java
package com.nareshit.dao;

import org.hibernate.Transaction;

import org.hibernate.Session;

import com.nareshit.pojo.Account;

import com.nareshit.utility.HibernateUtility;
```

```java
public class AccountDaoImpl  implements AccountDao{
    public int createAccount(Account acc){
        int id=0;
        Session session=HibernateUtility.getSession();
        if(session!=null){
            Transaction tx=session.beginTransaction();
            id=(Integer)session.save(acc);
            tx.commit();
            session.close();
        }//end of if block
        return id;
    }//end of createAccount(-) method
    public void updateAccountHolderName(int accNum,String newName){
        Session session=HibernateUtility.getSession();
        if(session!=null){
            Account acc=(Account)session.get(Account.class,accNum);
            if(acc!=null){
                acc.setName(newName);
                //update the record
                session.update(acc);
                session.beginTransaction().commit();
                System.out.println(accNum+" : Record Updated Successfully");
                session.close();
            }
            else{
                System.out.println("Account Not Found ");
            }
        }//end of outer if-block
    }//end of updateAccountHolderName(-) method
    public void deleteAccount(int accNum){
```

```java
                Session session=HibernateUtility.getSession();

                if(session!=null){

                        Account acc=(Account)session.get(Account.class,accNum);

                        if(acc!=null){

                                //delete the record

                                session.delete(acc);

                                session.beginTransaction().commit();

                                System.out.println(accNum+" : Record deleted Successfully");

                                session.close();

                        }

                        else{

                                System.out.println("Account Not Found ");

                        }

                }//end of Outer if-block

        }//end of deleteAccount(-) method

        public Account searchAccount(int accNum){

                Account acc=null;

                Session session=HibernateUtility.getSession();

                 if(session!=null){

                        acc=(Account)session.get(Account.class,accNum);

                 }

                return acc;

        }//end of searchAccount(-) method

}
```

Test.java

```java
package com.nareshit.client;
import com.nareshit.dao.AccountDao;
import com.nareshit.dao.AccountDaoImpl;
import com.nareshit.pojo.Account;
public class Test {
        public static void main(String[] args){
                AccountDao accDao=new AccountDaoImpl();
                System.out.println("Testing of createAccount(-) :");
```

```
//create the account obj
Account acc=new Account();
acc.setName("sathish hib");
acc.setBalance(99999999);
int id=accDao.createAccount(acc);
if(id>0){
        System.out.println("User Account created With Accnum :"+id);
}
else{
        System.out.println("User Account Not created ");
}
System.out.println("===============");
System.out.println("Testing of updateAccountHolderName(1002,sathish) :");
accDao.updateAccountHolderName(1002,"sathish");
System.out.println("===============");
System.out.println("Testing of deleteAccount(1003) ");
accDao.deleteAccount(1003);
System.out.println("===============");
System.out.println("Testing of searchAccount(1001) ");
Account account=accDao.searchAccount(1001);
if(account!=null){
        System.out.println(account);
}
else{
        System.out.println("Account Not Found ");
}
    }
}
```

Annotations Introduction

you have seen how Hibernate uses XML mapping file for the transformation of data

from POJO to database tables and vice versa.

Hibernate annotations is the new way to define mappings without a XML file.

You can use annotations in addition to (or) as a replacement of XML mapping metadata.

The important annotations are

@Entity

@Table

@Id

@GeneratedValue

@Column

With the help of the above annotations programmer's can easily

develop the meta-data information of their entity

@Entity,@Table,@Id,@Column,@GeneratedValue Annotations are present in javax.persistence package

These annotations are given by JPA(Java Persistence API). JPA is an ORM specification , Hibernate provides an implementations for JPA .

You use JPA annotations/API, by including Hibernate jars in your classpath.Hibernate also provided own annotations that annotations are present in org.hibernate.annotations package

public interface org.hiberate.annotations.@Entity extends javax.persistence.@Entity{

}

if we are using annotations from javax.persistence package You can use the ejb3-persistence.jar file in the classpath.

(in hibernate 3.x s/w this jar file is availiable in required folder)

Example :

@Entity //makes class as pojo class

@Table(name="emp_details") //maps with the db-table

public class Employee{

@Id //makes no as identiy filed

@Column(name="empNo") //maps the empNo property with empNo column of dbtable

private int empNo;

@Column(name="name",length=20)

private String name;

@Column(name="salary",length=10)

private double salary;

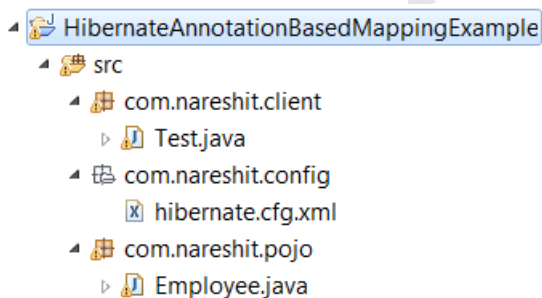**//required setters and getters**

}

Attributes of @Column Annotation

1 name (optional): the column name (default to the property name)

2 unique (optional): set a unique constraint on this column or not (default false)

3. nullable (optional): set the column as nullable (default true).

4. insertable (optional): whether or not the column will be part of the insert statement (default true)

5. updatable (optional): whether or not the column will be part of the update statement (default true)

6 columnDefinition (optional): override the sql DDL fragment for this particular column (non portable)

1. table (optional): define the targeted table (default primary table)

2. length (optional): column length

3. precision (optional): column decimal precision (default 0)

4. scale (optional): column decimal scale if useful (default 0)

- HibernateAnnotationBasedMappingExample
  - src
    - com.nareshit.client
      - Test.java
    - com.nareshit.config
      - hibernate.cfg.xml
    - com.nareshit.pojo
      - Employee.java

Employee.java

```java
package com.nareshit.pojo;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name="employee")
public class Employee {
@Id
@Column(name="empno")
private int empno;
@Column(name="deptno")
private int deptno;
@Column(name="empname",length=20)
private String name;
@Column(name="dptname",length=20)
private String deptname;
@Column(name="salary")
private double salary;

//Note:- @Id  and @Column annotation we can use variable level (OR) getterMethod

public int getEmpno() {
        return empno;
```

```
}
public void setEmpno(int empno) {
        this.empno = empno;
}
public int getDeptno() {
        return deptno;
}
public void setDeptno(int deptno) {
        this.deptno = deptno;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public String getDeptname() {
        return deptname;
}
public void setDeptname(String deptname) {
        this.deptname = deptname;
}
public double getSalary() {
        return salary;
}
public void setSalary(double salary) {
        this.salary = salary;
}
}
```

hibernate.cfg.xml

```xml
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver
</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE
</property>
<property name="connection.username">system</property>
<property name="connection.password">manager</property>
<property name="dialect">org.hibernate.dialect.Oracle9Dialect
</property>
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">update</property>
<mapping class="com.nareshit.pojo.Employee"/>
</session-factory>
</hibernate-configuration>
```

Note : if we are using annotation based mapping then in the hibernate configuration file we can use "mapping class " tag instead of " mapping resource" .

`<mapping class="com.nareshit.pojo.Employee"/>`

Test.java

```
package com.nareshit.client;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

import com.nareshit.pojo.Employee;

public class Test {

public static void main(String[] args){

Configuration cfg=new Configuration();

  cfg.configure("/com/nareshit/config/hibernate.cfg.xml");

 SessionFactory factory=cfg.buildSessionFactory();

 Session session=factory.openSession();

Employee emp=new Employee();

emp.setEmpno(2001);

emp.setName("Venky");

emp.setSalary(16000.0);

emp.setDeptno(12);

emp.setDeptname("IT");

Integer id=(Integer)session.save(emp);

 System.out.println("Employee inserted with  :"+id);

 }

}
```

Bulk Operations In Hibernate :-

While working with Hibernate ,Operations on Object's can be done in Two way's

1) Single row operations

2) Bulk Operations

In the case of Single row operations , The operations will be effected on a single row of a Database.

In the Case of Bulk operations, The operations will be effected on multiple rows at a time on Database.

Hibernate Framework Has Provided The Following 3 Techniques

To perform Bulk operations

1) HQL(Hibernate Query Language)

2) Criteria API

3) Native SQL

HQL (Hibernate Query Language):-

- Till Now we have done the operations on single object(single row),but with the HQL we will performs updates,deletes,selects on multiple rows of data(multiple objects) at a time

- To write complicated queries(or) multiple where conditions(or) to write where conditions on other than primary key columns (or) to work on multiple records we use the HQL features.

- We can perform both select and non-select operations on a database by using HQL.

- An object oriented form of SQL is called HQL.

- HQL syntax is very much similar to SQL syntax.

- HQL queries are formed by using entities and their properties,

- SQL quries are formed by using db tables and their columns. so SQL is a database dependent and HQL is a database Independent

Example:-

SQL Query

SQL:- select * from Employee ;

HQL Query

HQL:- select e from com.nareshit.pojo.Employee as e;
(or)
HQL :- from com.nareshit.pojo.Employee ;
(or)
HQL :- from com.nareshit.pojo.Employee as e ;

Here e is an identification variable denoting Employee Entity. by using 'as' construct we can create an identification variable but using of 'as' construct is an optional

- HQL and SQL keywords are not case sensitive.(Select,DeLete,FrOM)
- IN SQL DB table columns and Dbtablename is not case-sensitive.
- In HQL Entity name And Entity class variables are case-sensitive,
- HQL queries database independent queries. (HQL queries internally converted into d/b specific SQL queires using Dialect class).
- we can execute aggregate functions (MAX(),MIN(),AVG()...etc) using HQL.
- To write very complicated queries (or) which are not supported by HQL , we have to use Native SQL.Native SQL queries are database dependent queries. Maximum try to avoid using Native SQL.
- If we want to write the queries(HQL (or) Native SQL) declaratively(in mapping file(or) in annotations) we can use Named Queries.

Steps to work with HQL:-

Step 1: write the HQL query as per requirement.

Step 2: create an org.hibernate.Query object by passing HQL query.
To do this we can call createQuery(String hql) method from Session interface.
public Query createQuery(String hql)

Example :
Query query = session.createQuery(hqlquery);

Note :-
session.getNamedQuery(String namedQueryName) method is used to create An object of Query for a named Hql query string defined in the mapping file.
(We will learn about the Named Queries later in this chapter) .

Step 3 : if the Query contains parameters, set those values

Step 4 : execute query object.
query.list(); (OR) query.iterate() -->[for executing select queries]
query.executeUpdate(); →[for executing UPDATE/DELETE queries]

Query:-

Query is an interface,its implemented class as QueryImpl.

Query is an object oriented representation of Hibernate Query. The object of Query can be obtained by calling the session.createQuery(String hql) (OR)

session.getNamedQuery(String namedQueryName) method.

The Query interface provides many methods. Below are the commonly used methods:

1. **public int executeUpdate()** is used to execute the update or delete query.
2. **public List list()** returns the results of select query as a list.
3. **public Query setFirstResult(int rowno)** specifies the row number from where record will be retrieved.
4. **public Query setMaxResult(int rowno)** specifies the no. of records to be retrieved from the relation (table).

5. **public Query setParameter(int position, Object value)** it sets the value to the JDBC style query parameter.
6. **public Query setParameter(String name, Object value)** it sets the value to a named query parameter.

Differences between list() method iterate() method

1) list()-->directly returns the list data structure having selected records in the form of hb pojo class obj.
   iterate()-->returns Iterator obj pointing into list data structure .
2) list()-->generates single select sql query to select the records from the table.
   iterate()-->generates multiple select sql quereis to select the recods from db table.

Here first select query always returns identity values.

3) In the case of list()The returned HBpojo class obj will be intialized with the recods.irrespective of the obj utilization it is good in standalone applications.

In the case of iterate() The returned HBpojo class obj will be intialized with records on demand basis that means lazy loading of records is having . good in web-app that contains multiple layers.

Note:
When HQL select query selects all the column values of dbtable recods then the generated list data structure contains HB persistence class objects as element values by representing the records that are selected.

Example :


```
package com.nareshit.pojo;
public class Employee{
private int empNo;
private String name;
private double salary;
//required setters and getters
}
```

Employee Table

Empno	name	Salary
1001	ramu	5000.0
1002	raju	6000.0
1003	ajay	5500.0

HQL Query :

```
String hql = "FROM com.nareshit.pojo.Employee";
Query query = session.createQuery(hql);
List<Employee> results = query.list();
```

list data structure contains HB persistence class objects as element values by representing the records that are selected.

If HQL select Query is selecting specific column values of a table then the generated list data structure contains java.lang.Object[] array as an element values by representing records that are selected.

```
package com.nareshit.pojo;
public class Employee{                          Employee Table
private int empNo;
private String name;
private double salary;
//required setters and getters
}
```
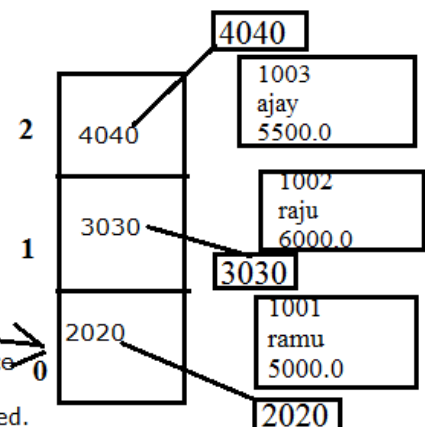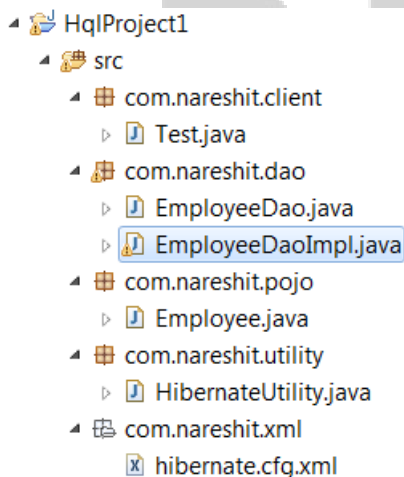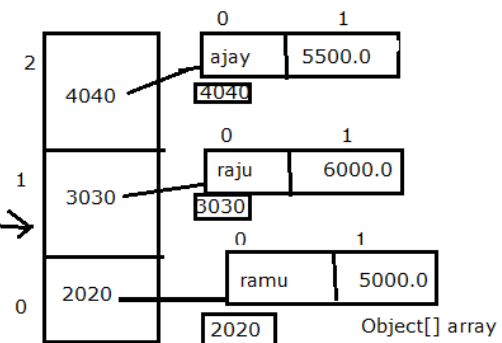
Empno	name	Salary
1001	ramu	5000.0
1002	raju	6000.0
1003	ajay	5500.0

HQL Query :
**String hql="select e.name,e.salary from com.nareshit.pojo.Employee as e";
Query query=session.createQuery(hql);
List<Object[]> results=query.list();**

list data structure contains java.lang.Object[] array as an element values by representing records that are selected

- HqlProject1
  - src
    - com.nareshit.client
      - Test.java
    - com.nareshit.dao
      - EmployeeDao.java
      - EmployeeDaoImpl.java
    - com.nareshit.pojo
      - Employee.java
    - com.nareshit.utility
      - HibernateUtility.java
    - com.nareshit.xml
      - hibernate.cfg.xml

Employee.java
package com.nareshit.pojo;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name="employee")
public class Employee {
        @Id
        @Column(name="empno")
        private int empno;
        @Column(name="name")
        private String name;

```java
        @Column(name="salary")
        private double salary;
        public int getEmpno() {
                return empno;
        }
        public void setEmpno(int empno) {
                this.empno = empno;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public double getSalary() {
                return salary;
        }
        public void setSalary(double salary) {
                this.salary = salary;
        }
}
```

EmployeeDAO.java

```java
package com.nareshit.dao;

import java.util.List;

import com.nareshit.pojo.Employee;

public interface EmployeeDAO{

    /**

     * this method is used to get the Employees By Name condition

     * @param name

     * @returns List<Employee>

     */

    List<Employee> getEmployeesByName(String name);

    /**

     * this method is used to get the Employees Name and salary

     * @param

     * @returns List<Object[]>
```

```
 */

 List<Object[]> getEmployees();

 /**

  * this method is used to update the EmployeeName

  * @param int ,String

  * @returns int

  */

 int updateEmployeeName(int empNo,String newName);

 /**

  * this method is used to delete the Employee

  * @param int

  * @returns int

  */

 int deleteEmployee(int empNo);

}
```

EmployeeDAOImpl.java

```
package com.nareshit.dao;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import com.nareshit.pojo.Employee;
import com.nareshit.utility.HibernateUtility;
public class EmployeeDAOImpl implements EmployeeDAO{
    public List<Employee> getEmployeesByName(String name) {
        List<Employee> results=null;
        Session session=HibernateUtility.getSession();
        if(session!=null){
    Query query=session.createQuery("from com.nareshit.pojo.Employee as e where e.name like
?");
            query.setParameter(0,name);
            results=query.list();
            session.close();
        }
        return results;
    }//end of getEmployeesByName
```

```java
        public List<Object[]> getEmployees() {
                List<Object[]> results=null;
                Session session=HibernateUtility.getSession();
                if(session!=null){
                        Query query=session.createQuery("select e.name,e.salary from
com.nareshit.pojo.Employee as e");
                        results=query.list();
                        session.close();
                }
                return results;
        }//end of getEmployees()

        public int updateEmployeeName(int empNo, String newName) {
                int count=0;
                Session session=HibernateUtility.getSession();
                if(session!=null){
                        Query query=session.createQuery("update com.nareshit.pojo.Employee as e
set e.name=? where e.empno=?");
                        query.setParameter(0,newName);
                        query.setParameter(1,empNo);
                        count=query.executeUpdate();
                        session.beginTransaction().commit();
                        session.close();

                }
                return count;
        }//end of updateEmployeeName(-,-)
        public int deleteEmployee(int empNo) {
                int count=0;
                Session session=HibernateUtility.getSession();
                if(session!=null){
                Query query=session.createQuery("delete from com.nareshit.pojo.Employee as e
where e.empno=?");
                        query.setParameter(0,empNo);
                        count=query.executeUpdate();
                        session.beginTransaction().commit();
                        session.close();
                }
                return count;
        }//end of deleteEmployee(-)
}
```

hibernate.cfg.xml

```xml
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">
oracle.jdbc.driver.OracleDriver</property>
```

```
<property name="connection.url">
jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">manager</property>
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">update</property>
<mapping class="com.nareshit.pojo.Employee"/>
</session-factory>
</hibernate-configuration>
```

HibernateUtility.java

```
package com.nareshit.utility;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HibernateUtility {
        private static SessionFactory factory;
        static{
                Configuration cfg=new Configuration();
                cfg.configure("/com/nareshit/xml/hibernate.cfg.xml");
                factory=cfg.buildSessionFactory();
        }
        public static Session getSession(){
                Session session=null;
                if(factory!=null){
                        session=factory.openSession();
                }
                return session;
        }
}
```

Test.java

```
package com.nareshit.client;
import java.util.List;
import com.nareshit.dao.EmployeeDao;
import com.nareshit.dao.EmployeeDaoImpl;
import com.nareshit.pojo.Employee;
public class Test {
        public static void main(String[] args){
                EmployeeDao edao=new EmployeeDaoImpl();
                System.out.println("Testing of getEmployeeByName ");
                List<Employee> list1=edao.getEmployeesByName("ramu");
                for(Employee emp:list1){
                        System.out.println("Emp No :"+emp.getEmpno());
                        System.out.println("Name :"+emp.getName());
                        System.out.println("Salary :"+emp.getSalary());
                }
                System.out.println("==========");
                System.out.println("Testing of getEmployees() ");
```

```
                    List<Object[]> list2=edao.getEmployees();
                    for(Object[] obj1:list2){
                            for(Object obj2:obj1){
                                    System.out.println(obj2);
                            }
                    }

                    System.out.println("==========");
                    System.out.println("Testing of updateEmployeeName ");
                    int updateCount=edao.updateEmployeeName(1001,"Ramu.Bandi");
                    System.out.println("Update Count :"+updateCount);
            System.out.println("==========");
                    System.out.println("Testing of deleteEmployee ");
                    int deleteCount=edao.deleteEmployee(1002);
                    System.out.println("Delete Count :"+deleteCount);
            }
}
```

When we are Preparing the HQL queries we can place 2 types of parameters

   ➢ Positional parameters(?)
   ➢ NamedParameters(:example)

These parameters helps the programmer to set input,conditional values to HQL queries as a java value. with out worrying about the syntax of HQL query.

To set the values to the above parametes we need to use setXxx(-,-) methods

(or) setParameter(-,-) methods

In Hibernate The positional parameters index is begins with '0' zero.

HQL Query with Positional Parameters:-

```
        Query query=session.createQuery("select e from com.nareshit.pojo.Employee as e where
        e.name like =? And e.salary >=?);

        q.setParameter(0,"ramu");

        q.setDouble(1,6000);

        //execute the Query

        List<Employee>  list=query.list();
```

If the multiple positional parameter are there in a single HQl query programmer may be confused while identifying the index's of parameters.

To overcome this problem it is recommended to work with named parameters.And Named Parameters will increase the readability of the query.

HQL Query with Named Parameter:-

Query query=session.createQuery("select e.empNo, e.name from com.nareshit.pojo.Employee as e where e.name like :name1 and e.salary >=:salary1";

//set values to Named parameters

query.setParameter("name1","ramu");

query.setDouble("salary1",7000.0);

List<Object[]> list=query.list();

Note:-

We can use both named parameters and positional parameters within the same query.But we should write positional parameters first then named parameters. i.e, once the named parameter if we are declaring is not allowed positional parameter.

String hql=" select e from com.nareshit.pojo.Employee as e where e.empNo=? and e.name like :name1";
Query query=session.createQuery(hql);
query.setParameter(0,1001);
query.setParameter("name1","ramu");
List list=query.list();

uniqueResult():-

if we are expecting only one record as the result of query executing then we will call

uniqueResult() method on query object . If query returns more than one record this method throws NonUniqueResultException.

Example 1 :   Table

empno	name	salary
1001	ramu	6000.0
1002	ramu	5800.0
1003	ajay	5500.0

String hql="from com.nareshit.pojo.Employee as e e.name like :name1";

Query query=session.createQuery(hql);

query.setParameter("name1","ramu");

Employee emp=(Employee)query.uniqueResult();

In the above example code query returns more than one record so we get Exception :org.hibernate.NonUniqueResultException :query did not return a unique result :2

If the query returns only one record uniqueResult() method executes the query and returns the resulted object.

Example 2:

String hql="from com.nareshit.pojo.Employee as e e.empNo =:empNo1";

Query query=session.createQuery(hql);

query.setParameter("empNo1",1001);

Employee emp=(Employee)query.uniqueResult();

System.out.println("Employee Details Are :");

System.out.println("EmpNo :"+emp.getEmpNo());

System.out.println("Name :"+emp.getName());

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

Example 3:

String hql="select e.name from com.nareshit.pojo.Employee as e where e.empNo=:empNo1";

Query query=session.createQuery(hql);

query.setParameter("empNo1",1001);

String empName=(String) query.uniqueResult();

System.out.println("EmpName :"+empName);

Example 4:

String hql="select e.name,e.salary from com.nareshit.pojo.Employee as e where e.empNo=:empNo1";

Query query=session.createQuery(hql);

query.setParameter("empNo1",1001);

Object[ ] obj=(Object[ ]) query.uniqueResult();

System.out.println("EmpName :"+obj[0]);

System.out.println("Emp Salary :"+obj[1]);

Note : Generally we will use this method where we should get only one record as a query result.

Aggregate Methods :

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL and following is the list of the available functions:

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

2	count(property name or *)	: The number of times a property occurs in the results
3	max(property name)	: The maximum value of the property values
4	min(property name)	: The minimum value of the property values
5	sum(property name)	: The sum total of the property values

Example 1 :

```
Session session = sessionFactory.openSession();
String hql = "Select min(e.salary) FROM Employee as e";
Query query = session.createQuery(hql);
System.out.println("Sum of Min salary : " + query.uniqueResult());
```

Example 2 :

```
Session session = sessionFactory.openSession();
String hql = "Select max(e.salary) FROM Employee as e";
Query query = session.createQuery(hql);
System.out.println("Sum of Max salary : " + query.uniqueResult());
```

Example 3 : Multiple Aggregate functions

```
Session session=factory.openSession();
String hql="select max(salary),min(salary) from com.nareshit.pojo.Employee";
List<Object[]> list=session.createQuery(hql).list();
for(Object [] obj:list){
System.out.println("Max Salary :"+obj[0]);
System.out.println("Min Salary:"+obj[1]);}
```

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).

Note:-

By-default ORDER BY clause will take ascending order.
If we want to specify the order explictly,we can write the queries as follows
HQL Query :-   From com.nareshit.pojo.Employee e ORDER BY e.name ASC
HQL Query :-   From com.nareshit.pojo.Employee e ORDER BY e.name DESC

Following is the simple example of using ORDER BY clause:

```
String hql = "FROM com.nareshit.pojo.Employee e WHERE e.empNo >=? ORDER BY e.salary DESC";
Query query = session.createQuery(hql);
query.setParameter(0,1001);
List<Employee> results = query.list();
```
If you want 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 com.nareshit.pojo.Employee e WHERE e.empNo > = ? ORDER BY e.name DESC,
e.salary DESC ";
Query query = session.createQuery(hql);
```

Query.setParameter(0,1001);
List<Employee> results = query.list();

Hibernate Named Query :

If there are a lot of queries, then they will cause a code mess because all the queries will be scattered throughout the project. That's why Hibernate provides **Named Query** that we can define at a central location and use them anywhere in the code. We can created named queries for both HQL and Native SQL.

There are two ways to define the named query in hibernate:

- by annotation
- by mapping file.

There are two types of NamedQueries

1) HQL NamedQueries
2) NativeSQL NamedQueries

With the Named Queries we have the following Advantages :

1. Hibernate Named Query helps us in grouping queries at a central location rather than letting them scattered all over the code.

2. Hibernate Named Query syntax is checked when the hibernate session factory is created, thus making the application fail fast in case of any error in the named queries.

3. Hibernate Named Query is global, means once defined it can be used throughout the application.

4. One of the major disadvantage of Named query is that it's hard to debug, because we need to find out the location where it's defined.

HQL NamedQueries :

Writing of HQL query in HBM file (OR) inside the pojo class with annotations is called HQL Named Query for HQL named queries we use <query> tag in HBM file.

For HQl named queries we no need to map the results. Because we write HQL query on Entity name and property names.

In HBM file we can write the Named Query inside the <class > definition (OR) outside the <class> defination

If namedquery is defined inside class defination then we must be access that named query with fully qualified Name.

To get the HQLNamedQuery we can use the following method from session interface

public Query getNamedQuery(String namedquery)

Named query definition has two important attributes:

- **name**: The name of name query by which it will be located using hibernate session.
- **query**: Here you give the HQL statement to get executed in database.

Example1 :

In Employee.hbm.xml

```
<hibernate-mapping>
<class name="com.nareshit.pojo.Employee" table="employee">
<id  name="empNo" column="empNo">
<generator class="assigned"></generator>
</id>
<property name="name" column="name"></property>
<property name="salary" column="salary"></property>
</class>
<query name="searchAllEmployees"> from com.nareshit.pojo.Employee as e </query>
</hibernate-mapping>
```

Client App:

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

Query query=session.getNamedQuery("searchAllEmployees");

List<Employee> list=query.list();
for(Employee                                                                       emp:list){
System.out.println(emp.getName());
System.out.println(emp.getSalary());
}
```

Example2:

In Employee.hbm.xml

```xml
<hibernate-mapping>
<class name="com.nareshit.pojo.Employee" table="employee">
<id  name="empNo" column="empNo">
<generator class="assigned"></generator>
</id>
<property name="name" column="name"></property>
<property name="salary" column="salary"></property>
<query name="searchAllEmployees"> from com.nareshit.pojo.Employee as e </query>
</class>
</hibernate-mapping>
```

Client App:

```java
Session session=sessionFactory.openSession();

Query query=session.getNamedQuery("com.nareshit.pojo.searchAllEmployees");

List<Employee> list=query.list();
for(Employee                                                   emp:list){
System.out.println(emp.getName());
System.out.println(emp.getSalary());

}
```

The following Example shows HQLNamedQueries

```
HQLNamedQueriesAnnotationMappingExample
  src
    com.nareshit.client
      Test.java
    com.nareshit.config
      hibernate.cfg.xml
    com.nareshit.dao
      EmployeeDAO.java
    com.nareshit.pojo
      Employee.java
    com.nareshit.utility
      HibernateUtility.java
  JRE System Library [JavaSE-1.7]
```

Employee.java

```java
package com.nareshit.pojo;
import javax.persistence.Column;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import  javax.persistence.Entity;
```

```java
@NamedQueries({
@NamedQuery(
name="findEmployeesByName",
query=
"from com.nareshit.pojo.Employee e where e.name like :name1 Order by e.salary"
            ),
@NamedQuery(
name="deleteEmployees",
query="delete from com.nareshit.pojo.Employee e where e.deptname=?"
            )}
)
@Entity
@Table(name="employee")
public class Employee {
        @Id
private int empno;
private int deptno;
private String name,deptname;
@Column(name="salary")
private double salary;

public int getEmpno() {
        return empno;
}
public void setEmpno(int empno) {
        this.empno = empno;
}
public int getDeptno() {
        return deptno;
}
public void setDeptno(int deptno) {
        this.deptno = deptno;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public String getDeptname() {
        return deptname;
}
public void setDeptname(String deptname) {
        this.deptname = deptname;
}
public double getSalary() {
        return salary;
}
public void setSalary(double salary) {
```

```
            this.salary = salary;
}


}
```

EmployeeDAO.java

```
package com.nareshit.dao;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.nareshit.pojo.Employee;
import com.nareshit.utility.HibernateUtility;
public class EmployeeDAO {
public List<Employee> searchEmployeeByName(String empName){
Session session=HibernateUtility.getSession();
Query query=session.getNamedQuery("findEmployeesByName");
query.setParameter("name1","%"+empName+"%");
List<Employee> list=query.list();
session.close();
return list;
}
public void removeEmployee(String deptname){
Session session=HibernateUtility.getSession();
Transaction tx=session.beginTransaction();
Query query=session.getNamedQuery("deleteEmployees");
query.setParameter(0,deptname);
    int count=query.executeUpdate();
    tx.commit();
    if(count>0){
  System.out.println("Employee Records Deleted");
    }
    else{
  System.out.println("Employee Records Not Deleted");
    }
}
}
```

hibernate.cfg.xml

```xml
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">
oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">
jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">manager</property>
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
```

```xml
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">update</property>
<mapping class="com.nareshit.pojo.Employee"/>
</session-factory>
</hibernate-configuration>
```

HibernateUtility.java

```java
package com.nareshit.utility;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HibernateUtility {
      private static SessionFactory factory;
        static {
      Configuration cfg=new Configuration();
      cfg.configure("/com/nareshit/config/hibernate.cfg.xml");
      factory=cfg.buildSessionFactory();
    }
  public static Session getSession()
      {
      Session session=null;
      if(factory!=null){
       session=factory.openSession();
      }
    return session;
    }
}
```

Test.java

```java
package com.nareshit.client;
import java.util.List;
import com.nareshit.dao.EmployeeDAO;
import com.nareshit.pojo.Employee;
public class Test{
      public static void main(String[] args) {
            EmployeeDAO edao=new EmployeeDAO();
            System.out.println("Testing of searchEmployeeByName :");
            List<Employee> list=edao.searchEmployeeByName("u");
            for(Employee emp:list){
                  System.out.println("Employee : "+emp.getEmpno()+" Details");

      System.out.println(emp.getName()+"\n"+emp.getDeptname()+"\n"+emp.getSalary());

            }
            System.out.println("Testing of removeEmployee:");
            edao.removeEmployee("Sales");
      }
}
```

Joins In Hibernate

- We use join statements, to select the data from multiple tables of the database, when there exist relationship
- with joins, its possible to select data from multiple tables of the database by construction a single query

Hibernate supports 4 types of joins..

- Left Join
- Right Join
- Full Join
- Inner Join

the **DEFAULT** join in hibernate is Inner join

- Left join means, the objects from both sides of the join are selected and more objects from left side are selected, even though no equal objects are there at right side
- Right join means equal objects are selected from database and more objects are from right side of join are selected even though there is no equal objects are exist left side
- Full join means, both equal and un-equal objects from both sides of join are selected
- Inner join means only equal objects are selected and the remaining are discarded
- At the time of construction the join statements, we need to use the properties created in pojo class to apply relationship between the objects
- To construct a join statement, we use either HQL, or NativeSql

For Example :-

the following example show sto use Inner Join query

Pojo Classes

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee{
@Id
    @GeneratedValue(strategy =
GenerationType.IDENTITY)
    @Column(name = "emp_No")
private int empNo;
@Column(name = "name",length=15)
private String name;
@Column(name="salary")
private double salary;
@OneToOne(mappedBy = "employee")
 @Cascade(value
=org.hibernate.annotations.CascadeType.ALL)
private Address;
//required setters and getters

}
```

```
public class Address {
 @Id
 @Column(name = "emp_No", unique = true, nullable
= false)
   @GeneratedValue(generator = "myGenerator")
   @GenericGenerator(name = "myGenerator",
strategy = "foreign",
     parameters = { @Parameter(name = "property",
value = "employee") })
   private int id;

@Column(name = "zipcode")
   private String zipcode;

  @Column(name = "city")
   private String city;
  @Column(name = "state")
   private String state;
   @OneToOne
   @PrimaryKeyJoinColumn
   private Employee employee;
//required setters and getters
}
```

Employee

emp_no (pk)	name	salary
1001	sathish	15000.0
1002	MVR	16000.0
1003	Hari	17000.0

Address

emp_no (pk)	city	state	zipcode
1001	hyd	TS	500018
1002	hyd	TS	500018
1003	hyd	TS	500018

DATABASE TABLES

Hibernate HQL Join Query Example

```
//join example

Query query = session.createQuery("select e.name, a.city from
Employee e INNER JOIN e.address a");

    List<Object[]> list = query.list();

    for(Object[] obj : list){

        System.out.println("Name :"+obj[0]);

        System.out.println("city :"+obj[1]);

    }
```

Pagination in Hibernate :

Pagination is the very common problem for the most of the Enterprise applications.When we are retrieving thousands of records from the database, it is not good idea to retrieve all the records at the same time. So, we have to implement some sort of *pagination* concept in your application to restrict the number of rows to be fetched from the database.

Pagination can be implemented in Two ways

    1. Front-End Pagination   2. Back-End Pagination

1. Front-End Pagination

In front-end pagination ,we will hit the database only once, and get the results and store them in the memory(session/application scope) , when user navigating

From one page to another page we retrieve the data from memory but not hitting the database again and again.

With this style of pagination we can achieve easy navigation ,fast loading of the data but we have the below problems

    i)   It takes more memory to hold results

    II)   If the data is updated in the database by other application,we cannot get the updated data because we are getting the data from memory instead of database.

Note : This type of pagination is suggestible if the number of resulted records are limited and the data is consistent data.

2. Back-End Pagination

In Back-end pagination ,we will hit the database again and again ,get's requested page details from database.

With this style of pagination we can achieve easy navigation, fast loading of data, takes memory to hold the result data And it will get always updated data.

But it gives less performance compared when compared with Front-end pagination.

Note : This type of pagination is suggestible ,if number of resulted records are unlimited and data is not consistent data.

To apply the pagination Hibernate provides two methods in org.hibernate.Query interface

public Query setFirstResult(int firstResult);→ It takes the argument which specifies from where record has to take.

public Query setMaxResults(int maxResults); → It takes the argument which specifies how many records has to take.

Example :-

```
String hql="from com.nareshit.pojo.Employee as e";
Query query=session.createQuery(hql);
query.setFirstResult(2); //Means starts from 3 rd record
query.setMaxResults(4);//Means No.of records 4 to be displayed
List<Employee> list=query.list();
for(Employee emp:list){
System.out.println("Emp No :"+emp.getEmpNo());
System.out.println("Emp Name :"+emp.getName());
System.out.println("Salary :"+emp.getSalary());
}
```

When we are using pagination ,In web page we are responsible to display page numbers ,When the user click on some page number ,we need to send the request to server,and get's corresponding page results .To do all these things we need to implement some logic.

The following Example showing Hibernate Pagination

Screen 1:

Welcome to Naresh it

EmployeeDetails

Screen 2:

Employee Details		
Employee Id	**Employee Name**	**Employee Salary**
1001	ajay	8000.0
1002	vijay	12000.0
1003	ramu	13000.0
1004	raju	32000.0

1 2 3

```
▲ 🗂 HibernatePaginationExample
   ▷ 🔊 JAX-WS Web Services
   ▷ 🔟 Deployment Descriptor: HibernatePaginationExample
   ▲ 🍃 Java Resources
      ▲ 🍃 src
         ▲ 🔣 com.nareshit.config
              🗴 hibernate.cfg.xml
         ▲ 🌐 com.nareshit.controller
            ▷ 🗾 EmployeeController.java
         ▲ 🌐 com.nareshit.dao
            ▷ 🗾 EmployeeDAO.java
            ▷ 🗾 EmployeeDAOImpl.java
         ▲ 🌐 com.nareshit.dao.factory
            ▷ 🗾 DAOFactory.java
         ▲ 🌐 com.nareshit.pojo
            ▷ 🗾 Employee.java
         ▲ 🌐 com.nareshit.service
            ▷ 🗾 EmployeeService.java
            ▷ 🗾 EmployeeServiceImpl.java
         ▲ 🌐 com.nareshit.util
            ▷ 🗾 HibernateUtility.java
      ▲ 📂 WebContent       ▲ 📂 WEB-INF
                              ▷ 📂 lib
                                 🗴 web.xml
                              📄 employeeDetails.jsp
                           📄 index.jsp
```

index.jsp

```html
<html><head><center><h3>Welcome to Naresh it</h3></center>
</head>
<body><br/><hr/><br/>
<center>
<a href="employeesDetails.do">EmployeeDetails</a>
</center>
</body>
</html>
```

web.xml

```xml
<web-app>
        <servlet>
                <servlet-name>employeeController</servlet-name>
                <servlet-class>com.nareshit.controller.EmployeeController
                </servlet-class>
        </servlet>
        <servlet-mapping>
                <servlet-name>employeeController</servlet-name>
                <url-pattern>*.do</url-pattern>
        </servlet-mapping>
        <welcome-file-list>
                <welcome-file>index.jsp</welcome-file>
        </welcome-file-list>
</web-app>
```

EmployeeController.java

```java
package com.nareshit.controller;
import java.io.IOException;
import java.util.List;
import javax.servlet.*;
import javax.servlet.http.*;
import com.nareshit.pojo.Employee;
import com.nareshit.service.*;
public class EmployeeController extends HttpServlet {
        private EmployeeService employeeService = null;
        public void init() {
                employeeService = new EmployeeServiceImpl();
        }
        protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                        throws ServletException, IOException {
                List<Employee> listOfEmployees = null;
                int pageIndex = 0;
                String startPageIndex = req.getParameter("pageIndex");
                if (startPageIndex == null) {
                        pageIndex = 1;
                } else {
                        pageIndex = Integer.parseInt(startPageIndex);
                }
                listOfEmployees= employeeService.getEmployees(pageIndex);
                long numberOfpages =employeeService.getNumberOfPages();
                req.setAttribute("employees", listOfEmployees);
                req.setAttribute("noOfPages", numberOfpages);
                RequestDispatcher rd =
req.getRequestDispatcher("employeeDetails.jsp");
                rd.forward(req, resp);
        }
}
```

EmployeeService.java

```java
package com.nareshit.service;
import java.util.List;
import com.nareshit.pojo.Employee;
public interface EmployeeService {
        public List<Employee> getEmployees(int pageIndex);
        public long getNumberOfPages();
        public int getStartPageIndex(int pageIndex);
        public long getTotalNumberOfEmployees();
}
```

EmployeeServiceImpl.java

```java
package com.nareshit.service;
import java.util.List;
import com.nareshit.dao.factory.DAOFactory;
import com.nareshit.pojo.Employee;
public class EmployeeServiceImpl implements EmployeeService {
        private final int numberOfRecordsPerPage =4;
        public int getStartPageIndex(int pageIndex){
                int startPageIndex = (pageIndex*numberOfRecordsPerPage)-
numberOfRecordsPerPage;
                return startPageIndex;
        }
        public long getNumberOfPages(){
                long totalNumberOfRecords = getTotalNumberOfEmployees();
                long numberOfPages = totalNumberOfRecords/numberOfRecordsPerPage;
                if(totalNumberOfRecords >(numberOfPages*numberOfRecordsPerPage)){
                        numberOfPages++;
                }
                return numberOfPages;
        }
        public List<Employee> getEmployees(int pageIndex) {
                List<Employee> listOfEmployees = null;

        listOfEmployees=DAOFactory.getEmployeeDAO().getEmployees(getStartPageIndex(pageInd
ex));
                return listOfEmployees;
        }
        public long getTotalNumberOfEmployees() {
                long count = DAOFactory.getEmployeeDAO().getTotalNumberRecords();
                return count;
        }

}
```

EmployeeDAO.java

```java
package com.nareshit.dao;
import java.util.List;
import com.nareshit.pojo.Employee;
public interface EmployeeDAO {
        public List<Employee> getEmployees(int pageIndex);
        public long getTotalNumberRecords();
```

}
EmployeeDAOImpl.java

```java
package com.nareshit.dao;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import com.nareshit.pojo.Employee;
import com.nareshit.util.HibernateUtility;
public class EmployeeDAOImpl implements EmployeeDAO {
        private static final int numberOfRecordsPerPage=4;
        public List<Employee> getEmployees(int pageIndex) {
                List<Employee> listOfEmployees = null;
                Session session = HibernateUtility.getSession();
                if(session!=null){
                        String hql = "FROM com.nareshit.pojo.Employee";
                        Query query = session.createQuery(hql);
                        query.setFirstResult(pageIndex);
                        query.setMaxResults(numberOfRecordsPerPage);
                        listOfEmployees=query.list();
                        HibernateUtility.closeSession(session);
                }
                return listOfEmployees;
        }

        public long getTotalNumberRecords() {
                long count=0L;
                Session session =       HibernateUtility.getSession();
                String hql = "SELECT COUNT(*) FROM com.nareshit.pojo.Employee";
                Query query = session.createQuery(hql);
                count= (Long) query.uniqueResult();
                HibernateUtility.closeSession(session);
                return count;
        }

}
```

Employee.java

```java
package com.nareshit.pojo;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="employee")
public class Employee {
        @Id
        @Column(name="empNo")
```

```java
        private int empNo;
        @Column(name="name",length=15)
        private String name;
        @Column(name="salary")
        private double salary;
        public int getEmpNo() {
                return empNo;
        }
        public void setEmpNo(int empNo) {
                this.empNo = empNo;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public double getSalary() {
                return salary;
        }
        public void setSalary(double salary) {
                this.salary = salary;
        }
}
```

DAOFactory.java

```java
package com.nareshit.dao.factory;
import com.nareshit.dao.EmployeeDAO;
import com.nareshit.dao.EmployeeDAOImpl;
public class DAOFactory {
        private static EmployeeDAO employeeDao;
        static{
                employeeDao=new EmployeeDAOImpl();
        }
        public static EmployeeDAO getEmployeeDAO(){
                return employeeDao;
        }
}
```

HibernateUtility.java

```java
package com.nareshit.util;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HibernateUtility {
        private static Configuration cfg;
        private static SessionFactory factory;
        static {
                cfg = new Configuration();
                cfg.configure("com/nareshit/config/hibernate.cfg.xml");
                factory = cfg.buildSessionFactory();
        }
```

```java
        public static Session getSession() {
                Session session = null;
                if (factory != null) {
                        session = factory.openSession();
                }
                return session;
        }

        public static void closeSession(Session session) {
                if (session != null) {
                        session.close();
                }
        }

}
```

hibernate.cfg.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
        <session-factory>
                <property
name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
                <property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
                <property name="connection.username">system</property>
                <property name="connection.password">manager</property>
                <property name="dialect">org.hibernate.dialect.Oracle10gDialect</property>
                <property name="show_sql">true</property>
                <property name="hbm2ddl.auto">update</property>
                <mapping class="com.nareshit.pojo.Employee" />
        </session-factory>
</hibernate-configuration>
```

employeeDetails.jsp

```jsp
<%@page import="java.util.*"%>
<%@page import="com.nareshit.pojo.*"%>
<html>
<title>Employee Details</title>
<body>
        <%!Employee emp = null;
        List<Employee> list = null;
        Long noOfPages = null;%>
        <table align="center" style="margin-top: 150px;">
                <tr>
                        <td colspan="3"><center
                                        style="font: bold; color: navy; color: white; background-color:
purple;">Employee
```

```jsp
                                    Details</center></td>
                </tr>
                <tr>
                        <th style="background: #03F; color: #FFF;">Employee Id</th>
                        <th style="background: #03F; color: #FFF;">Employee Name</th>
                        <th style="background: #03F; color: #FFF;">Employee Salary</th>
                </tr>
                <%
                        list = (List) request.getAttribute("employees");
                        if (list != null) {
                                for (Employee employee : list) {
                %>
                <tr>
                        <td style="background: #CCC;"><%=employee.getEmpNo()%></td>
                        <td style="background: #CCC;"><%=employee.getName()%></td>
                        <td style="background: #CCC;"><%=employee.getSalary()%></td>
                </tr>
                <%
                        }
                        } else {
                %>
                <center>
                        <%
                                out.println("Employees Not Found");
                        %>
                </center>
                <%
                        }
                %>
                <tr>
                        <td colspan="2">
                                <%
                                        noOfPages = (Long) request.getAttribute("noOfPages");
                                        if (list != null) {
                                                if (noOfPages != null) {
                                                        for (int i = 1; i <= noOfPages.longValue(); i++) {
                                                                String myurl =
"employeesDetails.do?pageIndex=" + i;
                                        %> <a href="<%=myurl%>"><%=i%></a> <%
        }
                }
        }
%>
                        </td>
                </tr>
        </table>
</body>
</html>
```

Generator's

Generator classes are used to generate the 'identifier' for a persistent object. i.e. While saving an object into the database, the generator Informs to the hibernate that, how the primary key value for the new record is going to generate.

Hibernate using different primary key generator algorithms, for each algorithm internally a java class is there for its implementation.

All Generator classes has to implement 'org.hibernate.id.ldentifierGenerator" interface, And has to override

generate (-,-) method. The logic to generate 'identifier' has to write in this method.

In built-in generator classes, identifier generation logic has implemented by using JDBC.

If we want to write the user defined generator class and it should implement 'org.hibernate.id.ldentifierGenerator' interface and has to override generate(-,-) method.

To configure generator class we can use <generator /> tag, which is sub element of <id/> tag <generator/> tag has one attribute called "class" with which we can be used tospecify generator class-name.

While configuring <generator /> tag in mapping file, if we need to pass anv parameters to generator class then we can use <param /> tag, which is the sub element of <generator/> tag,

Example: In HBM file
<hibernate-mapping>
<class name="ClassName" table="tableName" >
<id name="identifierVariableName" column="columnName"/>
<generator **class="generator-class-name">**
<param name="param-name">param-value </param>
</generator>
</id>
<property name="var2" column="column2"/>
</class>
</hibernate-mapping>

The following are the list of main generators we are using in the hibernate framework

1. sequence
2. assigned
3. increment
4. hilo
5. seqhilo
6. identity
7. native
8. uuid
9. guid
10. select
11. foregin

1. sequence (org.hibernate.id.SequenceGenerator)
This generator class is database dependent it means, we cannot use this generator class for all the databases, we should know whether the database supports sequence or not before we are working with it - It is not supporting by MYSQL db.

Here we write a sequence and it should be configured in HBM file and while persisting the object in the database, sequence is going to generate the identifier and it will assign to the Id property of persistent object, then it will store the persistent object into Database.

Steps:-

1. Create a Sequence in Database

SQL> create sequence empno_sequence start with 1000 increment by 1; to get the next value select empno_sequence.nextval from dual

2. Configure Sequence in HBM File.

```
<hibernate-mapping>
<class name="com.nareshit.pojo.Employee"  table="employee">
<id  name="empno">
<generator class="sequence">
<param name="sequence">Empno_SEQUENCE</param>
</generator>
</id>
<property name="name"></property>
<property name="salary"></property>
</class>
</hibernate-mapping>
```

Note: If we don't configure any sequence name then it will take default sequence name(HIBERNATE-SEQUENCE). But hibernate won't create the sequence, if already sequence is available with name "HIBERNATE-SEQUENCE", then it is used by hibernate. Otherwise hibernate raises the exception.

Note: But remember, if we enable hbrn2ddl.auto property in hibernate configuration file, then hibernate will create the database tables,sequences if they are not exist.

Note: It is not advisable to use the default sequence, always prefer to create a different sequence to each enitity seperately.

3. Create the entity and save it without assigning identifier.

1. **Session session** = **SessionUtil.getSession();**
2. **session. getTransaction ( ) .begin ( ) ;**
3. **Employee emp=new Employee();**
4. **emp. setName ("sathish") ;**
5. **emp. setSalary (5000) ;**
6. **Serializable id** = **session. save (emp) ;**
7. **System. out. println ("Employee is created with No : "+id) ;**
8. **session. getTransaction () . commit () ;**

Note: When we execute the above code, we can find the sequence execution query on the console, which is used to get identifier for the saving enitity.

Internal Code
```
public class SequenceGenerator implements PersistentIdentifierGenerator, Configurable {
public static final String SEQUENCE = "sequence";
public void configure(. . .){
sequenceName = PropertiesHelper.getString(SEQUENCE, params,"hibernate-sequence"); // default sequence name
```

---

parameters = params.getProperty(PARAMETERS);

public Serializable generate(. ..) {
Preparedstatement st =. . ..prepareSelectStatement(sql);
Resultset rs = st.executeQuery();
rs.next();
Serializable result = . . iterate results ...
return result;
}

Note: Now onwards for the following generator classes just give the HBM configuration, you can use the same entity saving logic(which we used in the above example as part of step-3) to test them.

2. Assigned (org.hibernate.id.Assigned)

This generator supports in all the databases
This is the default generator class used by the hibernate, if we do not specify <generator/> element under <id/> element, then hibernate by default assumes it as "assigned" generator class.

If generator class is assigned, then the programmer is responsible for assigning the identifier value to entity before saving into the database.

Example in code Mapping File:-
<id name=" empNo " column="empNo">
<generator **class="org.hibernate.id.Assigned" />**
</id >
(OR)
<id name=" empNo " column="EmpNO">
<generator **class="assigned" />**
</id >

Internal Code:
public class Assigned implements IdentifierGenerator,Configurable{
public Serializable generate(-,-){
final Serializable id = . . . it will gets the developer glven id.
if (id==null) {
throw new IdentifierGenerationException("ids for this class must be manually assigned before calling save(): " entityName);
 return id;
}

3. increment (org.hibernate.id.IncrementGenerator)

This generator supports in all the databases, so it is database independent generator class.
This generator is used for generating the id value for the new record by using the formula
Max of id value in Database + 1
If there is no record initially in the database, then for the first time this will saves primary key value as 1.

Example:-
<id name=" empNo " column="empNo">
<generator **class="increment"** />
</id >

(or)

<id name=" empNo " column="empNo">

<generator class=" **org.hibernate.id.lncrementGenerator** " /> </id>

4. High-low alg (org.hibernate.id.TableHiLoGenerator)

This generator is database independent
hilo uses a high/low algorithm to generate Identifiers.

This alg uses High-Low Alg to Generate identity values

- ➢ This alg uses specified table,related column value as the source of high value
- ➢ uses the max_low parameter value as the source of low value.
- ➢ uses following formula to generate identity value. (helper table column value)*(max_low value+1)

Example:-
create helper table having helper column with value.

SQL>create table mytable(mycolumn number(10));
SQL>insert into mytable value(10);
SQL>commit;

Ex:-
<id name="empNo" column="empNo">
<generator class="hilo">
<param name="table">mytable</param>
<param name="column">mycolumn</param>
<param name="max_lo">5</param>

</generator>
</id>

It is used the following formula to generate the Identifier (helpercolval)*(max_lo value+1)

- ⇨ (10)*(5+1)
     60-->11

     next

     11*(5+1)

==>66→ 12

If we are not configure helper table name ,helper column value and max_lo value.
By default hibernate will use helper table name as hibernate_unique_key and helper column value as next_hi
- And   for the first record, the id value will be inserted as 1
- for the second record the id value will be inserted as 32768
- for the next records the id value will be incremented by 32768 and will stores into the database (i mean adds to the previous)
- actually this hibernate stores the count of id values generated in a column of separated table, with name "**hibernate_unique_key**" by default with the column name "**next_hi**"

5. seqhilo(org.hibernate.id.SequenceHiLoGenerator):-

It is just like hilo generator class, But hilo generator stores its high value in table, where as seqhilo generator stores its high value in sequence.

This alg generates long,short,int type identity value by using given sequence generated value as high value& max_lo.

This alg using the following formula to generate the identity value:-

(sequence generated value)*(max_lo param value+1)

1*(5+1)=6

11*(5+1)=66

Example
```
<id name="empNo">
<generator class="seqhilo">
<param name="sequence">EmpNo_SEQUENCE</param>
<param name="max_lo">5</param>
</id>
```

6. identity(org.hibernate.id.IdentityGenerator)

This is database dependent, actually it's not working in oracle.
Identity columns are support by DB2, MYSQL, SQL SERVER, SYBASE and HYPERSYNCSQL databases.
This identity generator doesn't needs any parameters to pass
Syntax t o create identity columns in MYSQL database:

```
CREATE TABLE Employee(
EmpNO INT(10) NOT NULL AUTO-INCREMENT,
Name CHAR(2O),
salary  FLOAT,
PRIMARY KEY (EMPNO)
```

Example :
```
<id name=" empNo " column="EMPNO">
<generator class="identity" />
</id>
```

7. native:-

native is not having any generator class because,it uses internally
identity (or) sequence (or) hilo generator classes.

when we use this generator class, it first checks whether the database supports
identity or not, if the database not supporting the identity then it checks for sequence. if the database not supporting sequence then hilo will be used finally .

so the order will be..

identity
sequence
hilo

For example, if we are connecting with oracle, if we use generator class as native then it is equal to the generator class as sequence.

8. uuid (universalUniqueId)

uuid uses a 128- bit id algorithm to generate identifiers of type string.
uuid generated identifier is unique with in a network.
uuid algorithm generates identifier using IP address.
uuid algorithm codes identifier as a string(hexadecima1 digits) of length 32.
Generally uuld is used to generate passwords.

Example:-
```
<id name="empNo" column="empNO">
<generator class="uuid" >
 </generator>
</id>
```

9. guid(global unique identifier):
same as the uuid alg. It is database depdent.
it works on SQLServer and MYSQL.
It generates a string based identity value.

10. select:
select retrieves a primary key assigned by a database trigger by selecting the row by some unique key and retrieving the primary key value.

11. foreign:
This alg generates identity value for current obj by collecting the identity value of associated obj.
This alg is usefull in 1-1 association mapping.

User Defined Generator class
When we feel the existing generator classes are not fit for our requirement, then we will go for user defined generator class.

Steps to implement user defined generator class.

Step 1:-
Take any java class and implement org.hibernate.id .IdentifierGenerator and override generate() method.
In this method implement the identifier generation logic as per the requirement.

Step 2:-
Give the Generator class entity in the <generator> tag of HBM file.

Step1 : EmpNoGenerator.java

```java
package com.nareshit.id;
import java.io.Serializable;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.hibernate.HibernateException;
import org.hibernate.engine.spi.SessionImplementor;
import org.hibernate.id.IdentifierGenerator;
public class EmpNoGenerator  implements IdentifierGenerator{
public Serializable generate(SessionImplementor session, Object obj)
                          throws HibernateException {
     String empNo="Emp";
     try{
     Connection connection=session.connection();
     String query="select emp_no_sequence.nextval from dual";
     PreparedStatement  pst=connection.prepareStatement(query);
     ResultSet resultSet=pst.executeQuery();
     if(resultSet.next()){
     empNo=empNo+resultSet.getString(1);
               }
          }catch(SQLException se){
                  se.printStackTrace();
          }
          return empNo;
     }
}
```

In the above user defined Generator class i used database sequence to generate next value... And I got the connection from session object.

Step 2: Employee Mapping file :

```xml
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Employee" table="emp">
<id name="empNo">
<generator class="com.nareshit.id.EmpNoGenerator">
</generator>
</id>
<property name="name" length="12"></property>
<property name="salary" length="12"></property>
</class>
</hibernate-mapping>
```

Employee.java

```
package com.nareshit.pojo;
public class Employee {
private String empNo;
private String name;
private double salary;
//required setters and getters

}
```

Application Code :

```
Session session=sessionFactory.openSession();
Transaction tx=session.beginTransaction();
Employee emp=new Employee();
        emp.setName("ajay");
        emp.setSalary(22000.0);
    String id=(String)session.save(emp);
        tx.commit();
System.out.println("Emp Record Inserted with the following :"+id);
```

JPA Supported Identifier Generators

The @Id annotation lets you define which property is the identifier of your entity bean.

This property can be set by the application itself (OR) be generated by Hibernate(preferred). You can define the Identifier Generation strategy using @GeneratedValue annotation.

The @GeneratedValue annotation is used to specify the primary key generation

 strategy to use. If the strategy is not specified by default AUTO will be used.

Different GenerationType Strategies :

AUTO - either identity column, sequence or table depending on the underlying DB

it will work like native.

TABLE - hilo algorithm

IDENTITY - identity column

SEQUENCE - sequence

Note :-

Hibernate provides more id generators than the basic JPA

@GeneratedValue annotation will allow the following two attributes

strategy: It is used to set the generation strategy as defined in the javax.persistence.GenerationType

this can be :- AUTO, TABLE, SEQUENCE, IDENTITY

---

generator –It is used to specify a name of a generator the configurations are donated using the @TableGenerator, @SequenceGenerator,

 or @GenericGenerator (Hibernate specific)

Example 1:- With @GeneratedValue annotation :

Employee.java

```
package com.nareshit.pojo;
@Entity
@Table(name="employee")
public class Employee {
@Id
@GeneratedValue
@Column(name="eno")
private int empNo;
@Column(name="ename",length=12)
private String name;
@Column(name="esal")
private double salary;
//required setters and getters
}
```

Note : As we diid n't specify any strategy, by default it will take strategy=GenerationType.AUTO .As we know AUTO works like "native" generator,so it will take either "identity" (OR) "sequence"(OR)"hilo".

Example 2:- @GeneratedValue(strategy=GenerationType.AUTO)

Employee.java

```
package com.nareshit.pojo;
@Entity
@Table(name="employee")
public class Employee {
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="eno")
private int empNo;
@Column(name="ename",length=12)
private String name;
@Column(name="esal")
private double salary;
//required setters and getters
}
```

Example 3:- With @GeneratedValue(strategy=GenerationType.IDENTITY)

Note : This is not supported by Oracle,But supported by MySql database

Employee.java

```
package com.nareshit.pojo;
@Entity
@Table(name="employee")
public class Employee {
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
@Column(name="eno")
private int empNo;
@Column(name="ename",length=12)
private String name;
@Column(name="esal")
private double salary;
//required setters and getters
}
```

Example 4:- With @GeneratedValue(strategy=GenerationType.TABLE)

Employee.java

```
package com.nareshit.pojo;
@Entity
@Table(name="employee")
public class Employee {
@Id
@GeneratedValue(strategy=GenerationType.TABLE,generator="myGenerator")
@TableGenerator(name="myGenerator",initialValue=1000,allocationSize=1,table="PK_VALUE_TAB
",pkColumnName="PK_COLUMN",pkColumnValue="PK_VALUE",valueColumnName="PK_VALUE_C
OLUMN")
@Column(name="eno")
private int empNo;
@Column(name="ename",length=12)
private String name;
@Column(name="esal")
private double salary;
//required setters and getters
}
```

Hibernate Supported Identifier Generators :

To Integrate hibernate specific generators into JPA we can use either
@org.hibernate.annotations.GenericGenerator(OR)
@org.hibernate.annotations.GenericGenerators annotations

Example 1:- assigned(org.hibernate.id.Assigned)

Employee.java

```
package com.nareshit.pojo;
@Entity
@Table(name="employee")
public class Employee {
@Id
```

```
@GenericGenerator(name="myGenerator",strategy="assigned")
@GeneratedValue(generator="myGenerator")
@Column(name="eno")
private int empNo;
@Column(name="ename",length=12)
private String name;
@Column(name="esal")
private double salary;
//required setters and getters
}
```

Example 2:- increment(org.hibernate.id.IncrementGenerator)

Employee.java

```
package com.nareshit.pojo;
@Entity
@Table(name="employee")
public class Employee {
@Id
@GenericGenerator(name="myGenerator",strategy="increment")
@GeneratedValue(generator="myGenerator")
@Column(name="eno")
private int empNo;
@Column(name="ename",length=12)
private String name;
@Column(name="esal")
private double salary;
//required setters and getters
}
```

Example 3:- sequence(org.hibernate.id.SequenceGenerator)

Employee.java

```
package com.nareshit.pojo;
@Entity
@Table(name="employee")
public class Employee {
@Id
@GenericGenerator(name="myGenerator",strategy="sequence")
@GeneratedValue(generator="myGenerator")
@Column(name="eno")
private int empNo;
@Column(name="ename",length=12)
private String name;
@Column(name="esal")
private double salary;
//required setters and getters
}
```

Note : As there are no parameters so it will take default sequence name(Hibernate_Sequence)

Example 4:- sequence(org.hibernate.id.SequenceGenerator)

```
package com.nareshit.pojo;
@Entity
@Table(name="employee")
public class Employee {
@Id
@GenericGenerator(name="myGenerator",
strategy="sequence",
parameters={@Parameter(name="sequence",value="EMP_NO_SEQ")})
@GeneratedValue(generator="myGenerator")
@Column(name="eno")
private int empNo;
@Column(name="ename",length=12)
private String name;
@Column(name="esal")
private double salary;
//required setters and getters
}
```

Example 5:- UserDefined Generator (com.nareshit.id.EmpNoGenerator)

```
package com.nareshit.pojo;
@Entity
@Table(name="employee")
public class Employee {
@Id
@GenericGenerator(name="myGenerator",
strategy="com.nareshit.id.EmpNoGenerator")
@GeneratedValue(generator="myGenerator")
@Column(name="eno")
private int empNo;
@Column(name="ename",length=12)
private String name;
@Column(name="esal")
private double salary;
//required setters and getters
}
```

Composite ID:-

Process of defining primary key on more than one column is called composite id. Usage is combination of values should not be repeated

> Means

```
 ID1    ID2
1001    2001
```

Into the above columns it won't allow again 1001-2001 pair.but it allows 1001-2002,1003-2001.

It means combination should not be repeated.
Generally this concept implemented in link tables (many-many-relationship) we can implement this concept id in two ways.

1st way:-
Writing primary column related properties and other properties in the same entity.

2nd way:-
Writing primary column related properties in one entity class and other property in the other entity class.

Example Program
Files required....
Product.java (Pojo)
Test.java (main class)
hibernate.cfg.xml
Product.hbm.xml

Product.java
```
package com.nareshit;
public class Product implements java.io.Serializable{

private int productId;
private String productName;
private double price;

public void setProductId(int productId){
this.productId = productId;
}
public int getProductId(){
return productId;
}

public void setProductName(String productName){
this.proName = productName;
}
public String getProductName(){
return proName;
}

public void setPrice(double price){
this.price = price;
}
public double getPrice(){
return price;
}
}
```

hibernate.cfg.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
```

```
<session-factory>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver
</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">manager</property>

<property name="dialect">org.hibernate.dialect.OracleDialect</property>
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">update</property>

<mapping resource="Product.hbm.xml"></mapping>
</session-factory>
</hibernate-configuration>
```

Product.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
<class name="com.nareshit.Product" table="products">

<composite-id>
<key-property name="productId" column="pid"  />
<key-property name="productName" column="pname" length="10" />
</composite-id>

<property name="price"/>

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

Test.java

```
package com.nareshit;
import org.hibernate.*;
import org.hibernate.cfg.*;

public class Test{

  public static void main(String[] args) {

     Configuration cfg = new Configuration();
     cfg.configure("hibernate.cfg.xml");

     SessionFactory factory = cfg.buildSessionFactory();
     Session session = factory.openSession();

     Product p=new Product();

     p.setProductId(101);
```

```
            p.setProductName("keyboard");
            p.setPrice(250);

        Transaction tx=session.beginTransaction();
         session.save(p);
         System.out.println("Object Loaded successfully.....!!");
         tx.commit();

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

Note:-

- See Product.java pojo class, in line number 3 i have implemented the **java.io.Serializable**
- **hibernate.cfg.xml** is normal as previous programs, something like FirstHibernate program
- come to **Product.hbm.xml**, see line number **9**-**12,** this time we are using one new element<**composite-id**>
- Actually if we have a single primary key, we need to use <id> element, but this time we have multiple primary keys, so we need to use this new element <composite-id>

Persistent class Mapping :-

Configurations will do in hibernate mapping file are called   as Object Relational mapping configurations. we can perform the following ORM configurations.

1. Basic ORM(in all the above examples we are using Basic ORM)
2. Component Mapping
3. Inheritance Mapping
4. Collection Mapping
5. Association Mapping

Component-mapping:-

The property of hibernate persistence class that is mapped with a single column of database table is called as simple property.

Example :-

<property name="empName" column="EMP_Name"/>

The property of hibernate persistence class that is mapped with multiple columns of database table is called as component property. To use component - mapping we can work with <component> tag.

Example :-

<component name="address" class="com.nareshit.pojo.Address">
<property name="h_No" column="h_No" length="15"/>

<property name="city" column="city"/>
<property name="state" column="state"/>
<property name="country" column="country"/>

</component>
A component property holds represents multiple column values of a record. The data type of component property   is always an user-defined data types.

To configure simple property in hibernate mapping   file we can use <property> tag. Similarly to configure component property in hibernate mapping use <component> tag.
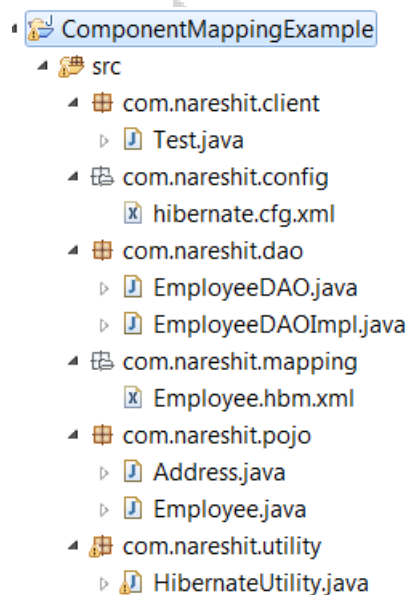
To demonstrate the component type we map Address as value type and Employee as Persistence class.

In the Below Example Employee persistence class with four properties➔one of them being of component type. i.e.Address class type and other three properties are empNo,name,salary.

 Address component type is implemented representing four properties such as h_No,state,city,country.

Example : Employee Table

EMPNO	NAME	SALARY	H_NO	CITY	STATE	COUNTRY
1001	ramu	-	3-2-1/a	hyd	TS	INDIA

- ComponentMappingExample
  - src
    - com.nareshit.client
      - Test.java
    - com.nareshit.config
      - hibernate.cfg.xml
    - com.nareshit.dao
      - EmployeeDAO.java
      - EmployeeDAOImpl.java
    - com.nareshit.mapping
      - Employee.hbm.xml
    - com.nareshit.pojo
      - Address.java
      - Employee.java
    - com.nareshit.utility
      - HibernateUtility.java

Address.java

package com.nareshit.pojo;

public class Address {
private String h_No,city,state,country;
public Address(){
}
public Address(String h_No,String city,String state,String country){
        this.h_No=h_No;
        this.state=state;
        this.city=city;
        this.country=country;
}
//required setters and getters

}
Employee.java

```java
package com.nareshit.pojo;
public class Employee {
private int empNo;
private String name;
private double salary;
private Address address;

//required setters and getters

}
```

Employee.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.nareshit.pojo">
<class name="Employee" table="employee">
<id name="empNo">
<generator class="increment"/>
</id>
<property name="name" length="15"/>
<property name="salary" length="10"/>
<component name="address" class="Address">
<property name="h_No"  column="h_No" length="15"/>
<property name="city"  column="city" length="15"/>
<property name="state" column="state" length="15"/>
<property name="country" column="country" length="15"/>
</component>
</class>
</hibernate-mapping>
```

hibernate.cfg.xml

same as the above applications

HibernateUtility.java

Same as the above applications

EmployeeDAO.java

```java
package com.nareshit.dao;
import com.nareshit.pojo.Employee;
public interface EmployeeDAO {
        public  void createEmployee(Employee emp);
        public  void getEmployee(int empno);
}
```
EmployeeDAOImpl.java

```java
package com.nareshit.dao;
```

```java
import org.hibernate.Session;
import com.nareshit.pojo.Employee;
import com.nareshit.utility.HibernateUtility;
public class EmployeeDAOImpl implements EmployeeDAO {
 public void createEmployee(Employee emp) {
Session session=HibernateUtility.getSession();
if(session!=null){
Integer id=(Integer)session.save(emp);
session.beginTransaction().commit();
System.out.println("Employee inserted with Id :"+id);
session.close();
               }
               }//end of createEmployee()
 public void getEmployee(int empno) {
Session session=HibernateUtility.getSession();
if(session!=null){
Employee emp=(Employee)session.get(Employee.class,empno);
System.out.println("Emp No : "+emp.getEmpNo());
System.out.println("Name :"+emp.getName());
System.out.println("Salary :"+emp.getSalary());
System.out.println("H_No :"+emp.getAddress().getH_No());
System.out.println("City :"+emp.getAddress().getCity());
System.out.println("State :"+emp.getAddress().getState());
System.out.println("Country :"+emp.getAddress().getCountry());
        }
}
}
```

Test.java

```java
package com.nareshit.client;
import com.nareshit.dao.EmployeeDAO;
import com.nareshit.dao.EmployeeDAOImpl;
import com.nareshit.pojo.Address;
import com.nareshit.pojo.Employee;
public class Test {
        public static void main(String[] args) {
        EmployeeDAO empDao=new EmployeeDAOImpl();
        Address address=new Address("3-2-1/a","hyd","TS","INDIA");
        Employee emp=new Employee();
        emp.setName("ramu");
        emp.setSalary(8000.0);
        emp.setAddress(address);
        System.out.println("Testing of createEmployee(-)");
        empDao.createEmployee(emp);
        System.out.println("Testing of getEmployee(1001)");
        empDao.getEmployee(1001);
        }

}
```

Inheritance Mapping :

Hibernate supports three different approaches of Inheritance hierarchy mapping:

1. Table Per Class Hierarchy
2. Table per subclass
3. Table per Concrete class

Now, let us discuss each of these approaches.

Table Per Class Hierarchy

In this approach all the Hibernate persistence classes of Inheritance will use single database table to store and and manage their data.

In this approach database table column have special column called discriminator column & this column contains some logical names which indicates each records in this table inserted by using specific Hibernate persistence class of Inheritance Hierarchy.

This approach is easier to use and efficient compared to the other two approaches.

But this approach suffers from one major problem the columns for properties declared by sub classes must be declared to accept null values, that is, these columns cannot be declared with not null constraint .This limitation may cause serious problems for data integrity.
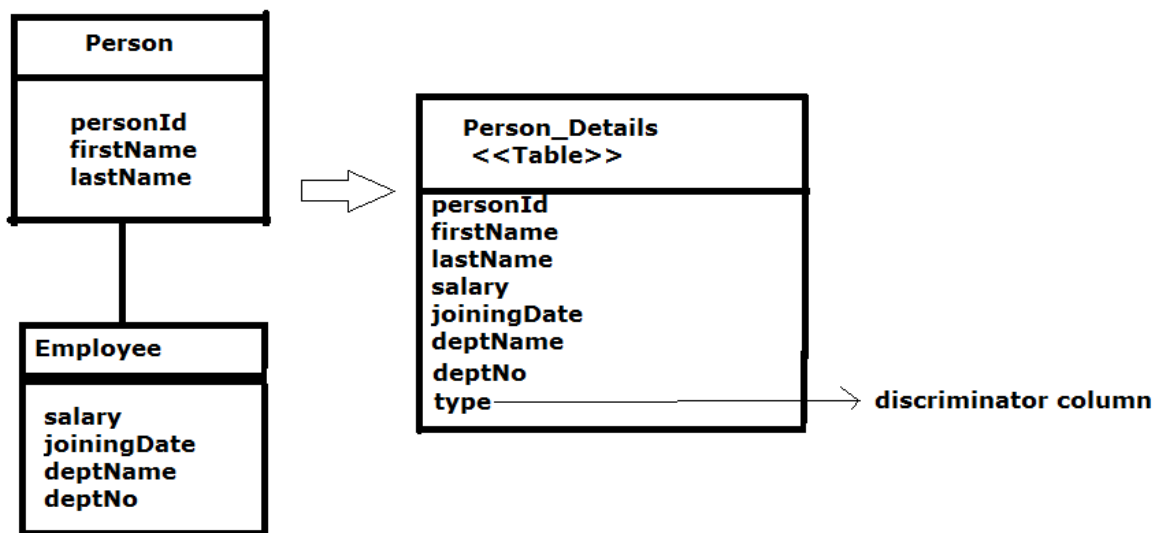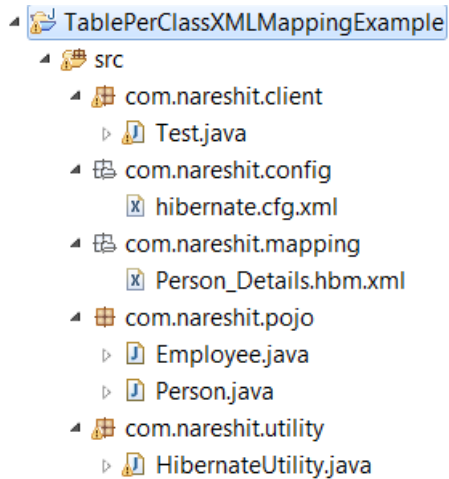

Table Per Class Hierarchy mapping

    <discriminator> tag is used to define the discriminator column.

    In this approach sub classes will be configured by using <subclass> tag .

SQL QUERY :

create table Person_Details (Person_ID number(10,0) not null, usertype varchar2(255 char) not null, firstName varchar2(15 char), lastName varchar2(15 char), salary double precision, deptNo number(10,0), deptName varchar2(20 char), joiningDate timestamp, primary key (Person_ID))

Example 1 :

- TablePerClassXMLMappingExample
  - src
    - com.nareshit.client
      - Test.java
    - com.nareshit.config
      - hibernate.cfg.xml
    - com.nareshit.mapping
      - Person_Details.hbm.xml
    - com.nareshit.pojo
      - Employee.java
      - Person.java
    - com.nareshit.utility
      - HibernateUtility.java

Person.java

```java
package com.nareshit.pojo;
public class Person {
private int personId;
private String firstName,lastName;
public int getPersonId() {
        return personId;
}
public void setPersonId(int personId) {
        this.personId = personId;
}
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;
}

}
```

Employee.java

```java
package com.nareshit.pojo;

import java.util.Date;

public class Employee extends Person {
private double salary;
private Date joiningDate;
private int deptNo;
private String deptName;
```

```java
public double getSalary() {
        return salary;
}
public void setSalary(double salary) {
        this.salary = salary;
}
public Date getJoiningDate() {
        return joiningDate;
}
public void setJoiningDate(Date joiningDate) {
        this.joiningDate = joiningDate;
}
public int getDeptNo() {
        return deptNo;
}
public void setDeptNo(int deptNo) {
        this.deptNo = deptNo;
}
public String getDeptName() {
        return deptName;
}
public void setDeptName(String deptName) {
        this.deptName = deptName;
}
}
```

Person _Details.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.nareshit.pojo">
<class name="Person" table="Person_Details" discriminator-value="person" >
<id name="personId" column="Person_ID">
<generator class="increment" / ></id>
<discriminator column="usertype" type="string" />
<property name="firstName"   length="15"/>
<property name="lastName"  length="15"/>
<subclass name="Employee"  extends="Person" discriminator-value="emp">
<property name="salary"  length="10"/>
<property name="deptNo"  length="10"/>
<property name="deptName"  length="20"/>
<property name="joiningDate"  type="java.util.Date" />
</subclass>
</class>
</hibernate-mapping>
```

hibernate.cfg.xml

same as the above applications

HibernateUtility.java

Same as the applications

Test.java

```java
package com.nareshit.client;
import java.util.Date;
import org.hibernate.Session;
import com.nareshit.pojo.Employee;
import com.nareshit.utility.HibernateUtility;
public class Test {
public static void main(String[] args){
        Test test=new Test();
        Employee emp=new Employee();
        emp.setFirstName("ramu");
        emp.setLastName("A");
        emp.setSalary(12000.0);
        emp.setDeptName("Sales");
        emp.setDeptNo(12);
        emp.setJoiningDate(new Date("12/12/2012"));
        System.out.println("Testing of createEmployee(-) ");
        test.createEmployee(emp);
System.out.println("Testing of getEmployee(1) :");
        test.getEmployee(1);
}
public  void createEmployee(Employee emp){
        Session session=HibernateUtility.getSession();
        Integer id=(Integer)session.save(emp);
        session.beginTransaction().commit();
        System.out.println("Employee created with id : "+id);
}
public  void getEmployee(int empId){
Session session=HibernateUtility.getSession();
Employee emp=(Employee)session.get(Employee.class,empId);
System.out.println(emp.getFirstName());
System.out.println(emp.getLastName());
System.out.println(emp.getDeptName());
System.out.println(emp.getJoiningDate());
System.out.println(emp.getSalary());
}
}
```

Table   :

PERSON_DETAILS

PERSON_ID	USERTYPE	FIRSTNAME	LASTNAME	SALARY	DEPTNO	DEPTNAME	JOININGDATE
1	emp	ramu	A	12000	12	Sales	12-DEC-12 12.00.00.000000 AM

The discriminator-value for Employee is defined *"emp"*, Thus, when Hibernate will persist the data for person or employee it will accordingly populate this value.

Example 2:-

- TablePerClassAnnotationMappingExample
  - src
    - com.nareshit.client
      - Test.java
    - com.nareshit.config
      - hibernate.cfg.xml
    - com.nareshit.pojo
      - Employee.java
      - Person.java
    - com.nareshit.utility
      - HibernateUtility.java

<u>Person.java</u>

```java
package com.nareshit.pojo;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
@Entity
@Table(name="Person_Details")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="usertype", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(value="Person")
public class Person {
        @Id
        @GeneratedValue(generator="myGenerator")
        @GenericGenerator(name="myGenerator",strategy="increment")
    private int personId;
     @Column(name="firtName",length=20)
     private String firstName;
  @Column(name="lastName",length=20)
   private String lastName;
    public int getPersonId() {
        return personId;
}
public void setPersonId(int personId) {
        this.personId = personId;
}
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;
}}
```

Employee.java

```java
package com.nareshit.pojo;
import java.util.Date;
import javax.persistence.*;
@Entity
@Table(name="Person_Details")
@DiscriminatorValue("emp")
public class Employee extends Person {
@Column(name="salary")
private double salary;
@Column(name="joiningDate")
private Date joiningDate;
@Column(name="deptNo")
private int deptNo;
@Column(name="deptName")
private String deptName;
public double getSalary() {
        return salary;
}
public void setSalary(double salary) {
        this.salary = salary;
}
public Date getJoiningDate() {
        return joiningDate;
}
public void setJoiningDate(Date joiningDate) {
        this.joiningDate = joiningDate;
}
public int getDeptNo() {
        return deptNo;
}
public void setDeptNo(int deptNo) {
        this.deptNo = deptNo;
}
public String getDeptName() {
        return deptName;
}
public void setDeptName(String deptName) {
        this.deptName = deptName;
}
}
```


hibernate.cfg.xml

```xml
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
```

```
<property name="connection.driver_class">
oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">
jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">manager</property>
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">update</property>
<mapping class="com.nareshit.pojo.Person"/>
<mapping class="com.nareshit.pojo.Employee"/>
</session-factory>
</hibernate-configuration>
```

HibernateUtility.java(same as the above application)

Test.java(same as the above application)

The Person class is the root of hierarchy. Hence we have used some annotations to make it as the root.

@Inheritance - Defines the inheritance strategy to be used for an entity class hierarchy. It is specified on the entity/persistence class that is the root of the entity class hierarchy.

@DiscriminatorColumn - Is used to define the discriminator column for the SINGLE_TABLE inheritance mapping strategies. The strategy and the discriminator column are only specified in the root of an entity class. +The name of the discriminator column defaults to "DTYPE" and the discriminator type to DiscriminatorType.STRING.

@DiscriminatorValue - Is used to specify the value of the discriminator column for entities of the given type. The Discriminator Value annotation can only be specified on a concrete entity class. If the Discriminator Value annotation is not specified and a discriminator column is used, a provider-specific function will be used to generate a value representing the entity type. If the Discriminator Type is STRING, the discriminator value default is the entity name.

Table Per Subclass:

In this approach every class of Inheritance hierarchy contains Its own database table. But the tables of Sub classs will maintain one-to-one relationship with table of Super class.

To configure Super class we can use <class> tag & and to configure Sub class we can use <joined-subclass> tags in Hibernate mapping file.

The <joined-subclass> tag is used to map the Sub class with Super class

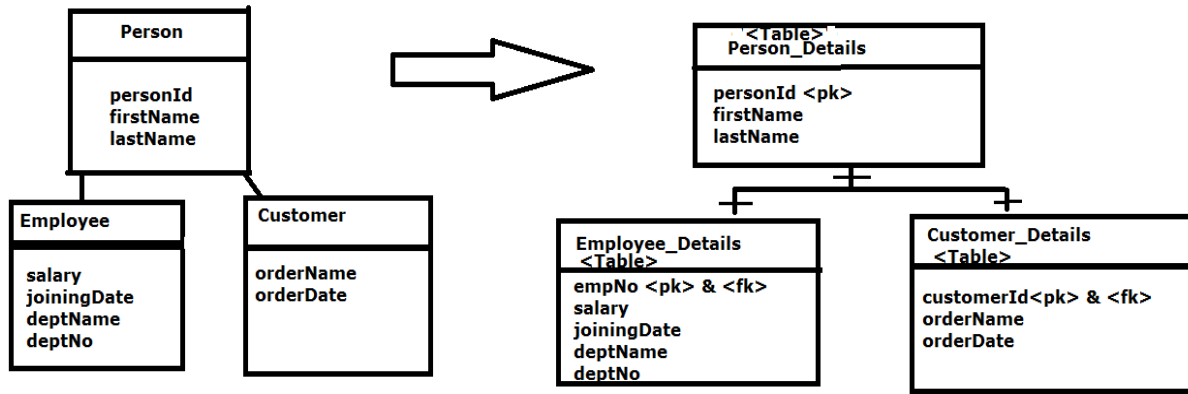Using the primary key and foreign key relation

Table Per subclass Hierarchy mapping

Note that a foreign key relationship exists between the subclass tables and super

class table. Thus the common data is stored in Person_Details table and subclass specific fields are stored in Employee_Details and Customer_Details tables.

SQL Scipt :

```
Create Database Tables

SQL> CREATE TABLE person_details (

    personid Number NOT NULL ,

    firstname VARCHAR(20) NOT NULL ,

    lastname VARCHAR(15) NOT NULL ,

    PRIMARY KEY (personid)

);

SQL>CREATE TABLE employee_details(

    empNo Number NOT NULL,

    joiningDate DATE NULL ,

    deptName VARCHAR(20) ,

    deptNo Number NOT NULL,

    salary Number NOT NULL,

    PRIMARY KEY (empNo),

    CONSTRAINT FK_PERSON1 FOREIGN KEY (empNo) REFERENCES person_details (personid)

);
```

```sql
SQL>CREATE TABLE customer_details(

   customerId Number NOT NULL,

   orderdate DATE NULL ,

   orderName VARCHAR(20) ,

   PRIMARY KEY (customerId),

   CONSTRAINT FK_PERSON2 FOREIGN KEY (customerId) REFERENCES person_details (personid)

)
```

Example :

```
TablePerSubclassExampleWithXMLMapping
  src
    com.nareshit.client
      Test.java
    com.nareshit.config
      hibernate.cfg.xml
    com.nareshit.mapping
      PersonDetails.hbm.xml
    com.nareshit.pojo
      Customer.java
      Employee.java
      Person.java
```

Person.java

```java
package com.nareshit.pojo;

public class Person {
private int personId;
private String firstName,lastName;
public int getPersonId() {
        return personId;
}
public void setPersonId(int personId) {
        this.personId = personId;
}
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;
```

```
}


}
```
Employee.java

```java
package com.nareshit.pojo;

import java.util.Date;

public class Employee  extends Person{
private int deptNo;
private double salary;
private String deptName;
private Date joiningDate;
public int getDeptNo() {
        return deptNo;
}
public void setDeptNo(int deptNo) {
        this.deptNo = deptNo;
}
public double getSalary() {
        return salary;
}
public void setSalary(double salary) {
        this.salary = salary;
}
public String getDeptName() {
        return deptName;
}
public void setDeptName(String deptName) {
        this.deptName = deptName;
}
public Date getJoiningDate() {
        return joiningDate;
}
public void setJoiningDate(Date joiningDate) {
        this.joiningDate = joiningDate;
}
}
```

Customer.java

```java
package com.nareshit.pojo;

import java.util.Date;

public class Customer extends Person{
private String orderName;
private Date orderDate;
public String getOrderName() {
```

```java
        return orderName;
    }
    public void setOrderName(String orderName) {
        this.orderName = orderName;
    }
    public Date getOrderDate() {
        return orderDate;
    }
    public void setOrderDate(Date orderDate) {
        this.orderDate = orderDate;
    }
}
```

PersonDetails.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.nareshit.pojo">
<class name="Person" table="Person_Details">
<id name="personId">
<generator class="increment"/>
</id>
<property name="firstName" length="15"/>
<property name="lastName" length="15"/>
<joined-subclass name="Employee"
table="Employee_Details">
<key column="empNo"/>
<property name="deptNo" length="10"/>
<property name="deptName" length="15"/>
<property name="salary" length="15"/>
<property name="joiningDate" />
</joined-subclass>
<joined-subclass name="Customer"
 table="Customer_Details">
<key column="customerId" />
<property name="orderName" length="15"/>
<property name="orderDate"  length="15"/>
</joined-subclass>
</class>
</hibernate-mapping>
```

The **joined-subclass** subelement of class, specifies the subclass. The **key** subelement of joined-subclass is used to generate the foreign key in the subclass mapped table. This foreign key will be associated with the primary key of parent class mapped table.

hibernate.cfg.xml

```xml
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
```

```
<session-factory>
<property name="connection.driver_class">
oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">
jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">manager</property>
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">update</property>
<mapping resource="com/nareshit/mapping/PersonDetails.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Test.java

```
package com.nareshit.client;
import java.util.Date;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import com.nareshit.pojo.Customer;
import com.nareshit.pojo.Employee;
public class Test {
public static void main(String[] args) {
        Test test=new Test();
        Configuration cfg=new Configuration();
        cfg.configure("/com/nareshit/config/hibernate.cfg.xml");
        SessionFactory factory=cfg.buildSessionFactory();
        Session session=factory.openSession();
System.out.println("Testing of Create Employee ");
test.createEmployee(session);
System.out.println("Testing of Create Customer ");
test.createCustomer(session);
System.out.println("Testing of Display Customer ");
test.displayCustomer(session);
System.out.println("Testing of Display Employee ");
test.displayEmployee(session);
        }
public  void displayEmployee(Session session){
Employee emp=(Employee)session.get(Employee.class,new Integer(1));
     System.out.println(emp.getFirstName());
                System.out.println(emp.getDeptName());
                System.out.println(emp.getSalary());
        }
public  void displayCustomer(Session session)
{
        Customer c=(Customer)session.get(Customer.class,new Integer(2));
        System.out.println(c.getFirstName());
System.out.println(c.getOrderName());
```

```java
System.out.println(c.getOrderDate());
}
public  void createEmployee(Session session){
        Employee emp=new Employee();
        emp.setFirstName("Ramu");
        emp.setLastName("A");
    emp.setDeptNo(12);
        emp.setDeptName("sales");
        emp.setSalary(12000);
        emp.setJoiningDate(new Date("12/12/2012"));

        session.save(emp);
        session.beginTransaction().commit();
}
public void createCustomer(Session session){
        Transaction tx=session.beginTransaction();
        Customer c=new Customer();
        c.setFirstName("B");
        c.setLastName("Sathish");
        c.setOrderName("Keyboard");
        c.setOrderDate(new Date("11/12/2012"));
        session.save(c);
        tx.commit();
}
}
```

Example 2: With Annotation mapping

```
TablePerSubClassExampleWithAnnotationMapping
  src
    com.nareshit.client
      Test.java
    com.nareshit.pojo
      Customer.java
      Employee.java
      Person.java
    com.nareshit.xml
      hibernate.cfg.xml
```

Person.java

```java
package com.nareshit.pojo;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

import org.hibernate.annotations.GenericGenerator;

@Entity
@Table(name="person_details")
@Inheritance(strategy=InheritanceType.JOINED)
```

```java
public class Person {
       @Id
       @GenericGenerator(name="myGenerator",strategy="increment")
       @GeneratedValue(generator="myGenerator")
       private Integer personId;
       private String firstName,lastName;
       public Integer getPersonId() {
              return personId;
       }
       public void setPersonId(Integer personId) {
              this.personId = personId;
       }
       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;
       }
}
```

Employee.java

```java
package com.nareshit.pojo;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;
@Entity
@Table(name="employee_details")
@PrimaryKeyJoinColumn(name="empNo")
public class Employee extends Person {
       private Integer deptNo;
       private String deptName;
       private Double salary;
       private Date joiningDate;
       public Integer getDeptNo() {
              return deptNo;
       }
       public void setDeptNo(Integer deptNo) {
              this.deptNo = deptNo;
       }
       public String getDeptName() {
              return deptName;
       }
```

```java
        public void setDeptName(String deptName) {
                this.deptName = deptName;
        }
        public Double getSalary() {
                return salary;
        }
        public void setSalary(Double salary) {
                this.salary = salary;
        }
        public Date getJoiningDate() {
                return joiningDate;
        }
        public void setJoiningDate(Date joiningDate) {
                this.joiningDate = joiningDate;
        }
}
```

Customer.java

```java
package com.nareshit.pojo;

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;
@Entity
@Table(name="customer_details")
@PrimaryKeyJoinColumn(name="customerId")
public class Customer extends Person {
        private String orderName;
        private Date orderDate;
        public String getOrderName() {
                return orderName;
        }
        public void setOrderName(String orderName) {
                this.orderName = orderName;
        }
        public Date getOrderDate() {
                return orderDate;
        }
        public void setOrderDate(Date orderDate) {
                this.orderDate = orderDate;
        }
}
```

Both Employee and Customer classes are child of Person class. Thus while specifying the mappings, we used @PrimaryKeyJoinColumn to map it to parent table.

@PrimaryKeyJoinColumn – This annotation specifies a primary key column that is used as a foreign key to join to another table.

It is used to join the primary table of an entity subclass in the JOINED mapping strategy to the primary table of its superclass; it is used within a SecondaryTable annotation to join a secondary table to a primary table; and it may be used in a OneToOne mapping in which the primary key of the referencing entity is used as a foreign key to the referenced entity.

If no PrimaryKeyJoinColumn annotation is specified for a subclass in the JOINED mapping strategy, the foreign key columns are assumed to have the same names as the primary key columns of the primary table of the superclass

Test.java

```
package com.nareshit.client;
import java.util.Date;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import com.nareshit.pojo.Customer;
import com.nareshit.pojo.Employee;
public class Test {
public static void main(String[] args) {
        Test test=new Test();
        Configuration cfg=new Configuration();
        cfg.configure("/com/nareshit/config/hibernate.cfg.xml");
        SessionFactory factory=cfg.buildSessionFactory();
        Session session=factory.openSession();
System.out.println("Testing of Create Employee ");
test.createEmployee(session);
System.out.println("Testing of Create Customer ");
test.createCustomer(session);
System.out.println("Testing of Display Customer ");
test.displayCustomer(session);
System.out.println("Testing of Display Employee ");
test.displayEmployee(session);


        }
public  void displayEmployee(Session session){
Employee emp=(Employee)session.get(Employee.class,new Integer(1));
    System.out.println(emp.getFirstName());
            System.out.println(emp.getDeptName());
            System.out.println(emp.getSalary());
        }
public  void displayCustomer(Session session)
{
        Customer c=(Customer)session.get(Customer.class,new Integer(2));
        System.out.println(c.getFirstName());
System.out.println(c.getOrderName());
System.out.println(c.getOrderDate());
}
public  void createEmployee(Session session){
        Employee emp=new Employee();
```

```
        emp.setFirstName("Ramu");
        emp.setLastName("A");
    emp.setDeptNo(12);
        emp.setDeptName("sales");
        emp.setSalary(12000);
        emp.setJoiningDate(new Date("12/12/2012"));
        session.save(emp);
        session.beginTransaction().commit();
}
public void createCustomer(Session session){
        Transaction tx=session.beginTransaction();
        Customer c=new Customer();
        c.setFirstName("B");
        c.setLastName("Sathish");
        c.setOrderName("Keyboard");
        c.setOrderDate(new Date("11/12/2012"));
        session.save(c);
        tx.commit();
}
}
```

In Table per class hierarchy we design table with huge no.of columns. This is against of database designing. But In Table Per sub class approach

Multiple table will be there in shorter form & having the relationship along them .

so this is good database designing. In Real world the Regularly used Inheritance mapping approach is Table-per sub class.

In Table per sub class of Inheritance mapping each record of Sub tables maintains

One to one relationship with Super table.

Table per Concrete class:-

Table per Concrete class on one per pojo class basis.

In the case of Table Per Concrete class, there will be multiple 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

Association Mapping :

Instead of taking single -large table with huge amount of columns for a project. It is recommended to design multiple data base tables & and keep them in a relationship/association. so that we can work with those table individually.

Primary key, foreign keys are used to form the relationships between the tables in Database.

Types of Relationships in DataBase :

      1. one-to-one

      2. one-to -many

      3. many-to-one

      4. many-to-many

p.k ---> p.k   (one-to-one)

f.k(unique) ---> p.k (one-to-one)

p.k ----> f.k (one-to-many)

f.k-----> p.k (many-to-one)

table1---> linktable --->table2(many-to-many)

To implement the association mapping in hibernate-mapping file we can use the following tags.

<one-to-one>

<one-to-many>

<many-to-one>

<many-to-many>

ONE-TO-ONE:-

In this association one entity object exactly mapping with/associated with One object of another entity.

**An Example of this type of association is the Relationship between an Address & Person.**

In one-to- One relationship each Person should have an unique address. Hence we will need two tables namely PERSON and ADDRESS to create this relation.

In Hibernate, one-to-one relationship between entities can be created by 2 different techniques.

These techniques are:

1. Using shared primary key:

In this technique, hibernate will ensure that it will use a common primary key value in both the tables. This way primary key of entity PERSON can safely be assumed the primary key of entity ADDRESS also. The example demonstrated shared primary key. The relational model is shown below:
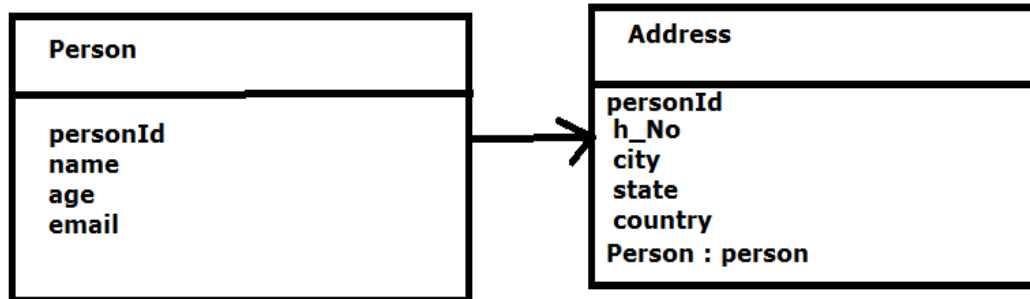
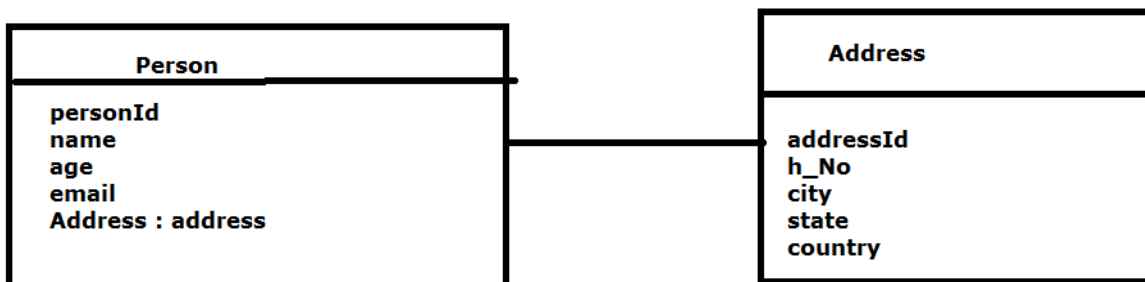one -to - one Releation class Diagram

Example Tables :

Person Table

personId (pk)	name	age	email
1001	ramu	23	ramu@gmail.com
1002	raju	22	raju@gmail.com

Address Table

personId (pk)	h_No	city	state	country
1001	2-31/a	hyd	TS	INDIA
1002	1-13/b	hyd	TS	INDIA

2. Using Foreign Key Association:

In this association, a foreign key column is created in the owner entity.

For example, if we make PERSON as owner entity, then a extra column "ADDRESS_ID" will be created in PERSON table. This column will store the foreign key for ADDRESS table. The relational model is shown below:

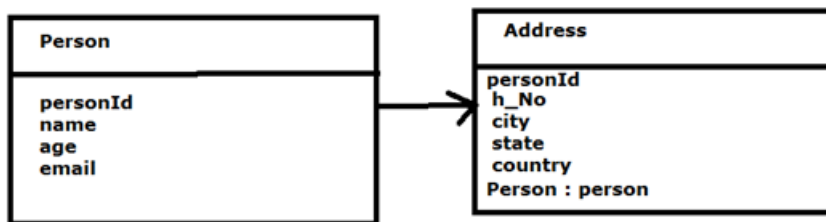one -to- one relation class Diagram

Example Tables :

Person Table

personId (pk)	name	age	email	address_Id (fk)(unique)
1001	ramu	23	ramu@gmail.com	2001
1002	raju	22	raju@gmail.com	2002

Address Table

addressId (pk)	h_No	city	state	country
2001	2-31/a	hyd	TS	INDIA
2002	1-12/b	hyd	TS	INDIA

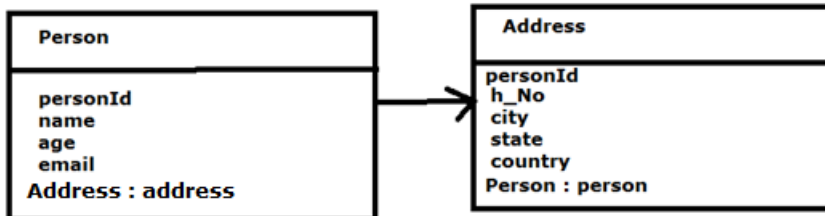The following examples demonstrated shared primary key. The relational model is shown below:

Example Table :

Person Table

personId (pk)	name	age	email
1001	ramu	23	ramu@gmail.com
1002	raju	22	raju@gmail.com

Address Table

personId (pk)	h_No	city	state	country
1001	2-31/a	hyd	TS	INDIA
1002	1-13/b	hyd	TS	INDIA

one -to - one Releation class Diagram

Uni Directional

one -to - one Releation class Diagram

Bi-Directional

One-to-One (pk-pk) UniDirectionalXmlMappingExample

SQL Script:

SQL>create table person (personId number(10,0) not null, name varchar2(12 char), age number(10,0), email varchar2(15 char), primary key (personId));

SQ>create table address (personId number(10,0) not null, h_No varchar2(20 char), city varchar2(20 char), state varchar2(20 char), country varchar2(20 char), primary key (personId));

- one-to-one(pk-to-pk)UniDirectionalXMLMappingExample
  - src
    - com.nareshit.client
      - Test.java
    - com.nareshit.config
      - hibernate.cfg.xml
    - com.nareshit.mapping
      - Address.hbm.xml
      - Person.hbm.xml
    - com.nareshit.pojo
      - Address.java
      - Person.java

Person.java

```java
package com.nareshit.pojo;
public class Person {
private int personId,age;
private String name,email;
public int getPersonId() {
        return personId;
}
public void setPersonId(int personId) {
        this.personId = personId;
}
public int getAge() {
        return age;
}
public void setAge(int age) {
        this.age = age;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public String getEmail() {
        return email;
}
public void setEmail(String email) {
        this.email = email;
}

}
```

Address.java

```java
package com.nareshit.pojo;
public class Address {
private int personId;
```

```java
private String h_No,city,state,country;
private Person person;
public int getPersonId() {
        return personId;
}
public void setPersonId(int personId) {
        this.personId = personId;
}
public String getH_No() {
        return h_No;
}
public void setH_No(String h_No) {
        this.h_No = h_No;
}
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 getCountry() {
        return country;
}
public void setCountry(String country) {
        this.country = country;
}
public Person getPerson() {
        return person;
}
public void setPerson(Person person) {
        this.person = person;
}
}
```

Person.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Person" table="person">
<id name="personId">
<generator class="increment"/>
</id>
<property name="name" length="12"/>
```

```
<property name="age" length="3"/>
<property name="email" length="15"/>
</class>
</hibernate-mapping>
```

Address.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Address" table="address">
<id name="personId">
<generator class="foreign">
<param name="property">person</param>
</generator></id>
<property name="h_No" length="20"/>
<property name="city" length="20"/>
<property name="state" length="20"/>
<property name="country" length="20"/>
<one-to-one name="person" class="com.nareshit.pojo.Person"/>
</class>
</hibernate-mapping>
```

Note:- In the above example code Address object identity value is nothing but associated person class object identity value. for this we need to use foreign algorthim.

Foreign alg collects identity value from associated object and makes it has the identity value of current object. only foreign alg is capable of Collecting the identity value from associated object and assigns as identity value of current object.
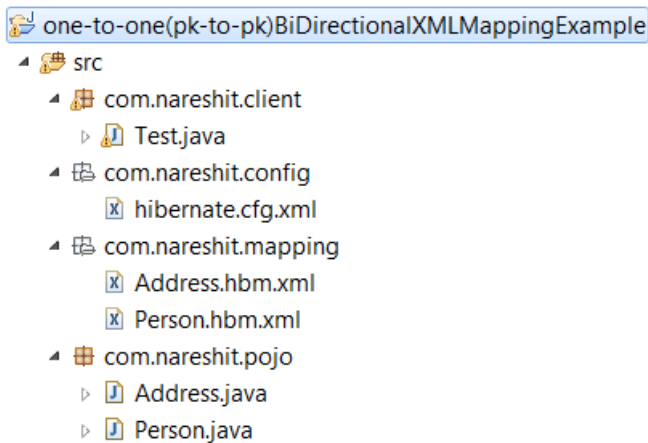
Test.java

```
package com.nareshit.client;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import com.nareshit.pojo.Address;
import com.nareshit.pojo.Person;
public class Test {
        public static void main(String[] args) {
                Configuration cfg=new Configuration();
                cfg.configure("/com/nareshit/config/hibernate.cfg.xml");
                SessionFactory factory=cfg.buildSessionFactory();
                Session session=factory.openSession();
                Transaction tx=session.beginTransaction();
                Person person=new Person();
                person.setName("ramu");
                person.setAge(23);
```

```
                person.setEmail("ramu@gmail.com");
                Address address=new Address();
                address.setH_No("2-31/a");
                address.setCity("hyd");
                address.setState("TS");
                address.setCountry("INDIA");
                address.setPerson(person);
                Integer id=(Integer)session.save(address);
                tx.commit();
                System.out.println("person created with Id :"+id);
        }
}
```

Example 2: One –to –One (pk-to-pk) BiDirectional XMLMappingExample

```
one-to-one(pk-to-pk)BiDirectionalXMLMappingExample
▲ 🏢 src
  ▲ 🎛 com.nareshit.client
    ▷ 📄 Test.java
  ▲ 🎛 com.nareshit.config
      🗙 hibernate.cfg.xml
  ▲ 🎛 com.nareshit.mapping
      🗙 Address.hbm.xml
      🗙 Person.hbm.xml
  ▲ 🏢 com.nareshit.pojo
    ▷ 📄 Address.java
    ▷ 📄 Person.java
```

Person.java

```
package com.nareshit.pojo;
public class Person {
private int personId,age;
private String name,email;
private Address address;

public Address getAddress() {
        return address;
}
public void setAddress(Address address) {
        this.address = address;
}
public int getPersonId() {
        return personId;
}
public void setPersonId(int personId) {
        this.personId = personId;
}
public int getAge() {
        return age;
}
public void setAge(int age) {
        this.age = age;
```

```java
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public String getEmail() {
        return email;
}
public void setEmail(String email) {
        this.email = email;
}


}
```

Address.java

```java
package com.nareshit.pojo;
public class Address {
private int personId;
private String h_No,city,state,country;
private Person person;
public int getPersonId() {
        return personId;
}
public void setPersonId(int personId) {
        this.personId = personId;
}
public String getH_No() {
        return h_No;
}
public void setH_No(String h_No) {
        this.h_No = h_No;
}
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 getCountry() {
        return country;
}
```

```java
public void setCountry(String country) {
        this.country = country;
}
public Person getPerson() {
        return person;
}
public void setPerson(Person person) {
        this.person = person;
}
}
```
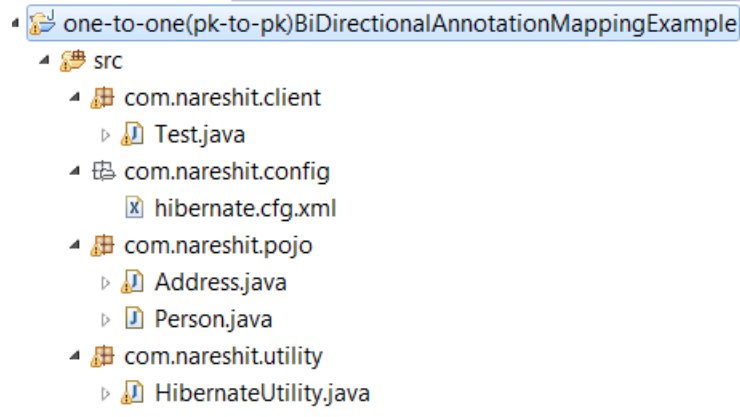
Person.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Person" table="person">
<id name="personId">
<generator class="increment"/>
</id>
<property name="name" length="12"/>
<property name="age" length="3"/>
<property name="email" length="15"/>
<one-to-one name="address"
class="com.nareshit.pojo.Address" />
</class>
</hibernate-mapping>
```

Address.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Address" table="address">
<id name="personId">
<generator class="foreign">
<param name="property">person</param>
</generator>
</id>
<property name="h_No" length="20"/>
<property name="city" length="20"/>
<property name="state" length="20"/>
<property name="country" length="20"/>
<one-to-one name="person" class="com.nareshit.pojo.Person" />
</class>
</hibernate-mapping>
```

hibernate.cfg.xml And Test.java files same as the

one-to-one(pk-pk)UniDirectionalXmlMappingExample

Example 3: One –to –One (pk - to-pk) Bidirectional Annotation Mapping Example

- one-to-one(pk-to-pk)BiDirectionalAnnotationMappingExample
  - src
    - com.nareshit.client
      - Test.java
    - com.nareshit.config
      - hibernate.cfg.xml
    - com.nareshit.pojo
      - Address.java
      - Person.java
    - com.nareshit.utility
      - HibernateUtility.java

Person.java

```java
package com.nareshit.pojo;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;
import org.hibernate.annotations.GenericGenerator;
@Entity
@Table(name="person")
public class Person {
        @Id
        @GeneratedValue(generator="myGenerator")
        @GenericGenerator(name = "myGenerator",          strategy = "increment")
        @Column(length=12)
private int personId;
private int age;
@Column(length=20)
private String name,email;
@OneToOne(cascade=CascadeType.ALL)
@PrimaryKeyJoinColumn
private Address address;
public Address getAddress() {
        return address;
}
public void setAddress(Address address) {
        this.address = address;
}
public int getPersonId() {
        return personId;
```

```java
}
public void setPersonId(int personId) {
        this.personId = personId;
}
public int getAge() {
        return age;
}
public void setAge(int age) {
        this.age = age;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public String getEmail() {
        return email;
}
public void setEmail(String email) {
        this.email = email;
}
}
```

Address.java

```java
package com.nareshit.pojo;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
import org.hibernate.annotations.Parameter;
@Entity
@Table(name="Address")
public class Address {
        @Id
        @GenericGenerator(name = "myGenerator",
        strategy = "foreign",
        parameters=@Parameter(name = "property",
        value = "person"))
        @GeneratedValue(generator="myGenerator")
private int personId;
        @Column(length=20)
private String h_No,city,state,country;
@OneToOne(mappedBy = "address",cascade=CascadeType.ALL)
private Person person;
public int getPersonId() {
        return personId;
}
public void setPersonId(int personId) {
        this.personId = personId;
}
```

```java
public String getH_No() {
        return h_No;
}
public void setH_No(String h_No) {
        this.h_No = h_No;
}
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 getCountry() {
        return country;
}
public void setCountry(String country) {
        this.country = country;
}
public Person getPerson() {
        return person;
}
public void setPerson(Person person) {
        this.person = person;
}
}
```

hibernate.cfg.xml

```xml
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
        <property name="connection.password">manager</property>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
                <property name="hbm2ddl.auto">update</property>
                <property name="show_sql">true</property>
                <mapping class="com.nareshit.pojo.Person"/>
<mapping class="com.nareshit.pojo.Address"/>
        </session-factory>
</hibernate-configuration>
```

HibernateUtility.java

```java
package com.nareshit.utility;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HibernateUtility {
private static SessionFactory factory;
static{
Configuration cfg=new Configuration();
cfg.configure("/com/nareshit/config/hibernate.cfg.xml");
factory=cfg.buildSessionFactory();
}
public static Session getSession(){
        Session session=null;
        if(factory!=null){
                session=factory.openSession();
        }
        return session;
}
}
```

Test.java

```java
package com.nareshit.client;
import java.util.Iterator;
import java.util.List;

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

import com.nareshit.pojo.Address;
import com.nareshit.pojo.Person;
import com.nareshit.utility.HibernateUtility;
public class Test {
        public static void main(String[] args) {
                Person person=new Person();
                person.setName("ramu");
                person.setAge(23);
                person.setEmail("ramu@gmail.com");
                Address address=new Address();
                address.setH_No("2-31/a");
                address.setCity("hyd");
                address.setState("TS");
                address.setCountry("INDIA");
Test test=new Test();
System.out.println("Testing of savePersonInfo(-,-)");
```

```java
Integer personId=test.savePersonInfo(person, address);
            System.out.println("Person saved :"+personId);
            System.out.println("Testing of listPersonInfo()");
            test.listPersonInfo();
    }
    public Integer savePersonInfo(Person person, Address address) {
            Session session = HibernateUtility.getSession();
    Integer personId = null;
    Transaction transaction = null;
    transaction = session.beginTransaction();
    try {
      address.setPerson(person);
     personId=(Integer) session.save(address);
      transaction.commit();
    } catch (HibernateException e) {
       transaction.rollback();
       e.printStackTrace();
    } finally {
       session.close();
    }
    return personId;
       }
       public void listPersonInfo() {
    Session session = HibernateUtility.getSession();
          try {
            List<Person>personList = session.createQuery("from com.nareshit.pojo.Person").list();
            for (Iterator<Person> iterator = personList.iterator(); iterator
                  .hasNext();) {
              Person person = (Person) iterator.next();
              System.out.println(person.getName());
              System.out.println(person.getEmail());
             System.out.println(person.getAddress().getH_No() + " "
                    + person.getAddress().getCity() + " "
                    + person.getAddress().getState());
            }
          } catch (HibernateException e) {

            e.printStackTrace();
          } finally {
            session.close();
          }
       }
}
```

One-to-One Using foreign key association :-

1. In this association, a foreign key column is created in the owner entity. For example, if we make PERSON as owner entity, then a extra column "ADDRESS_ID" will be created in PERSON table. This

column will store the foreign key for ADDRESS table. The relational model is shown below:

one -to- one relation class Diagram
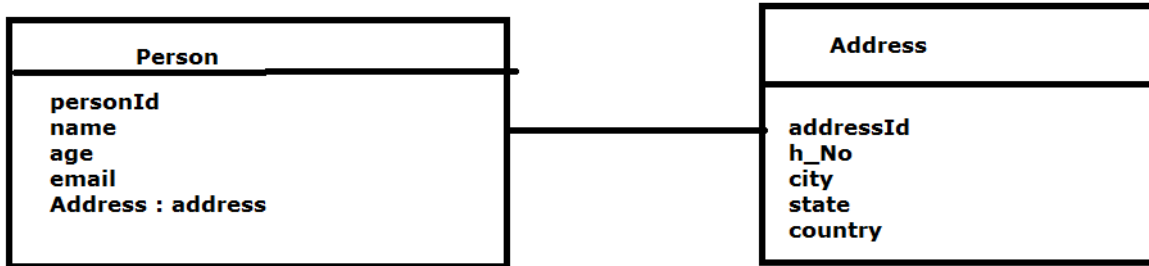
Example Tables :

Person Table

personId (pk)	name	age	email	address_Id (fk)(unique)
1001	ramu	23	ramu@gmail.com	2001
1002	raju	22	raju@gmail.com	2002

Address Table

addressId (pk)	h_No	city	state	country
2001	2-31/a	hyd	TS	INDIA
2002	1-12/b	hyd	TS	INDIA

OnetoOne(fk(unique)-pk)AnnotationMappingExample1
- src
  - com.nareshit.client
    - Test.java
  - com.nareshit.config
    - hibernate.cfg.xml
  - com.nareshit.dao
    - PersonDao.java
  - com.nareshit.pojo
    - Address.java
    - Person.java
  - com.nareshit.utility
    - HibernateUtility.java

Person.java

```
package com.nareshit.pojo;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
@Entity
@Table(name="person")
public class Person {
        @Id
        @GenericGenerator(name="myGenerator",strategy="increment")
        @GeneratedValue(generator="myGenerator")
        private int personId;
        @Column(length=12,name="name")
        private String personName;
        private byte age;
```

```java
        @Column(length=15,name="email")
        private String email;
@OneToOne(cascade=CascadeType.ALL)
@JoinColumn(name="address_Id",unique=true,nullable=false)
private Address address;
public Person(){
}
public Person(String personName, byte age, String email) {
                this.personName = personName;
                this.age = age;
                this.email = email;
        }
public Address getAddress() {
        return address;
}
public void setAddress(Address address) {
        this.address = address;
}        public int getPersonId() {
                return personId;
        }
        public void setPersonId(int personId) {
                this.personId = personId;
        }
public String getPersonName() {
                return personName;
        }
public void setPersonName(String personName) {
                this.personName = personName;

        public byte getAge() {
                return age;
        }
        public void setAge(byte age) {
                this.age = age;
        }
        public String getEmail() {
                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }
}
```

Address.java

```java
package com.nareshit.pojo;
import javax.persistence.*;
@Entity
@Table(name="Address")
public class Address {
```

```java
@Id
private int addressId;
@Column(length=15,name="h_No")
private String h_No;
@Column(length=20,name="city")
private String city;
@Column(length=20,name="state")
private String state;
@Column(length=20,name="country")
private String country;
public Address(){
//zero-arg constructor
}
public Address(int addressId,String h_No,String city,String state,String country){
this.addressId=addressId;
this.h_No=h_No;
this.city=city;
this.state=state;
this.country=country;
}
public int getAddressId() {
        return addressId;
}
public void setAddressId(int addressId) {
        this.addressId = addressId;
}
public String getH_No() {
        return h_No;
}
public void setH_No(String h_No) {
        this.h_No = h_No;
}
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 getCountry() {
        return country;
}
public void setCountry(String country) {
        this.country = country;
}}
```

PersonDao.java

```java
/**
 *
 */
package com.nareshit.dao;
import org.hibernate.Session;
import com.nareshit.pojo.Person;
import com.nareshit.utility.HibernateUtility;

/**
 * @author Sathish
 *
 */
public class PersonDao {
public void createPerson(Person person){

Session session=HibernateUtility.getSession();
if(session!=null){
   Integer personId=(Integer)session.save(person);
        session.beginTransaction().commit();
        System.out.println("person Id :"+personId);
        session.close();
}
}
public void getPersonDetails(int personId){
Session session=HibernateUtility.getSession();
Person person=(Person) session.get(Person.class,personId);
System.out.println("Person Details");
System.out.println("Name :"+person.getPersonName());
System.out.println("Email :"+person.getEmail());
System.out.println("Age :"+person.getAge());
System.out.println("City :"+person.getAddress().getCity());
System.out.println("State :"+person.getAddress().getState());
}
}
```

In one –to-one (fk- to -pk) association, refer the Address entity in Person class as follows:

1 @OneToOne

2 @JoinColumn(name="address_Id")

3 private Address address;

If no **@JoinColumn** is declared on the owner (Person) entity, the defaults apply. A join column(s) will be created in the owner entity table and its name will be the concatenation of the name of the relationship in the owner side, _ (underscore), and the name of the primary key column(s) in the owned side.

hibernate.cfg.xml

```xml
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">
oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">
jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">manager</property>
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">update</property>
<mapping class="com.nareshit.pojo.Person"/>
<mapping class="com.nareshit.pojo.Address"/>
</session-factory>
</hibernate-configuration>
```

Test.java

```java
/**
 *
 */

package com.nareshit.client;

import com.nareshit.dao.PersonDao;

import com.nareshit.pojo.Address;

import com.nareshit.pojo.Person;

/*
 * @author sathish
 *
 */

public class Test {

public static void main(String[] args){

PersonDao personDao=new PersonDao();

Person personDetails=

new Person("ramu",new Byte((byte) 22),"ramu@gmail.com");

Address address=new Address(2001,"2-31/a","hyd","TS","India");
```

personDetails.setAddress(address);

System.out.println("Testing of createPerson(-) ");

personDao.createPerson(personDetails);

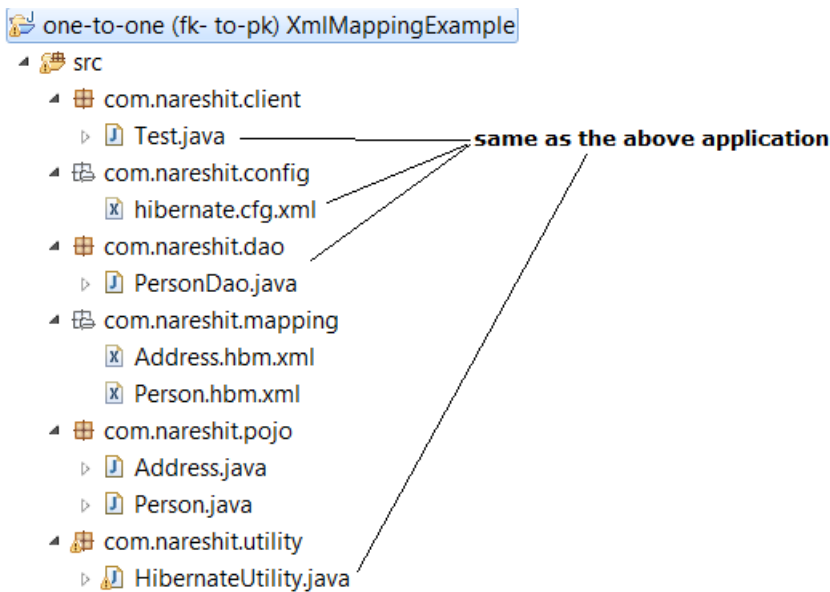System.out.println("Testing of getPersonDetails(1) : ");

personDao.getPersonDetails(1);

}

}

Example 2: one-to-one (fk-pk)Unidirectional XmlMappingExample

```
one-to-one (fk- to-pk) XmlMappingExample
  src
    com.nareshit.client
      Test.java ————————— same as the above application
    com.nareshit.config
      hibernate.cfg.xml
    com.nareshit.dao
      PersonDao.java
    com.nareshit.mapping
      Address.hbm.xml
      Person.hbm.xml
    com.nareshit.pojo
      Address.java
      Person.java
    com.nareshit.utility
      HibernateUtility.java
```

Person.java

package com.nareshit.pojo;

public class Person {
private int personId;
        private String personName;
        private byte age;
        private String email;
        private Address address;
        public Person(){
        }
        public Person(String personName,**byte** age,String email){
                this.personName=personName;
                this.age=age;
                this.email=email;
        }
        public int getPersonId() {
                return personId;
        }
        public void setPersonId(**int** personId) {

```java
                this.personId = personId;
        }
        public String getPersonName() {
                return personName;
        }
        public void setPersonName(String personName) {
                this.personName = personName;
        }
        public byte getAge() {
                return age;
        }
        public void setAge(byte age) {
                this.age = age;
        }
        public String getEmail() {
                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }
        public Address getAddress() {
                return address;
        }
        public void setAddress(Address address) {
                this.address = address;
        }
}
```

Address.java

```java
package com.nareshit.pojo;

public class Address {
private int addressId;
private String h_No;
private String city;
private String state;
private String country;
public Address(){
//zero-arg constructor
}
public Address(int addressId,String h_No,String city,String state,String country){
this.addressId=addressId;
this.h_No=h_No;
this.city=city;
this.state=state;
this.country=country;
}
public int getAddressId() {
        return addressId;
```

```java
}
public void setAddressId(int addressId) {
        this.addressId = addressId;
}
public String getH_No() {
        return h_No;
}
public void setH_No(String h_No) {
        this.h_No = h_No;
}
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 getCountry() {
        return country;
}
public void setCountry(String country) {
        this.country = country;
}
}
```

Person.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Person"
table="person">
<id name="personId">
<generator class="increment"></generator>
</id>
<property name="personName" column="name" length="12"/>
<property name="email" length="15"/>
<property name="age" length="3"/>
<many-to-one name="address" column="address_Id"
unique="true" cascade="all" not-null="true"
 class="com.nareshit.pojo.Address"/>
</class>
</hibernate-mapping>
```

In the above mapping file

The ***<many-to-one name="address" class="com.nareshit.pojo.Address" column="address_Id" unique="true" cascade="all" /> is the tag in the above hbm xml that configures the One To One relation from Person to Adddress.***

The column attribute describes the column name in the table mapping to this entity which is a reference key ferering to the ID of the other entity. That is in simple the FK column name.

Address.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Address"
table="address">
<id name="addressId">
</id>
<property name="h_No" length="15"/>
<property name="city" length="15"/>
<property name="state" length="15"/>
<property name="country" length="15"/>
</class>
</hibernate-mapping>
```

Many-to-one:-

This type of Association relates many objects of an Entity to one Object of another Entity. An Example of this type of association is the relationship b/w Employee and a Department.

For Example Multiple Employee are associated with one Dept.

To map many to one relation we can use <many-to-one> element

in hibernate mapping file of Employee persistence class.

Example in Employee mapping file:

```
<many-to-one name="dept"
class="com.nareshit.pojo.Department"
column="dept_no"/>
```

column attribute specifies the foreign key from the declaring

to the referring table.

Employee_Details

empNo (pk)	name	salary	dept_No (fk)
101	sathish	15000.0	12
102	ramesh	16000.0	12
103	raju	14000.0	13
104	ramu	14500.0	13
105	pavan	15500.0	12

Department_Details

deptNo (pk)	deptName	location
12	Sales	hyd
13	Admin	hyd

```
▲ 📂 Many-to-OneXMLMappingExample1
  ▲ 📂 src
    ▲ ⊞ com.nareshit.client
      ▷ 🔳 Test.java
    ▲ ⊞ com.nareshit.config
        🗷 hibernate.cfg.xml
    ▲ ⊞ com.nareshit.dao
      ▷ 🔳 DepartmentDAO.java
      ▷ 🔳 EmployeeDAO.java
    ▲ ⊞ com.nareshit.mapping
        🗷 Department.hbm.xml
        🗷 Employee.hbm.xml
    ▲ ⊞ com.nareshit.pojo
      ▷ 🔳 Department.java
      ▷ 🔳 Employee.java
    ▲ ⊞ com.nareshit.utility
      ▷ 🔳 HibernateUtility.java
```

Employee.java

```java
package com.nareshit.pojo;
public class Employee {
private int empNo;
private String name;
private double salary;
private Department dept;
public Employee(){
}
public Employee(String name,double salary){
        this.name=name;
        this.salary=salary;
}
public int getEmpNo() {
        return empNo;
}
public void setEmpNo(int empNo) {
        this.empNo = empNo;
}
public String getName() {
```

```java
        return name;
}
public void setName(String name) {
        this.name = name;
}
public double getSalary() {
        return salary;
}
public void setSalary(double salary) {
        this.salary = salary;
}
public Department getDept() {
        return dept;
}
public void setDept(Department dept) {
        this.dept = dept;
}
}
```

Department.java

```java
package com.nareshit.pojo;
public class Department {
private int deptNo;
private String deptName,location;
public Department(){
        }
public Department(int deptNo,String deptName,String location){
this.deptNo=deptNo;
this.deptName=deptName;
this.location=location;
}
public int getDeptNo() {
        return deptNo;
}
public void setDeptNo(int deptNo) {
        this.deptNo = deptNo;
}
public String getDeptName() {
        return deptName;
}
public void setDeptName(String deptName) {
        this.deptName = deptName;
}
public String getLocation() {
        return location;
}
public void setLocation(String location) {
        this.location = location;
}}
```

Employee.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Employee"
table="Employee_Details">
<id name="empNo" length="12">
<generator class="sequence">
<param name="sequence">EmpNo_Sequence</param>
</generator>
</id>
<property name="name" length="15"/>
<property name="salary" length="7"/>
<many-to-one name="dept"
class="com.nareshit.pojo.Department"
column="dept_no"  not-null="true" cascade="all"/>
</class>
</hibernate-mapping>
```

Department.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Department"
table="Department_Details">
<id name="deptNo">
</id>
<property name="deptName" length="15"/>
<property name="location" length="20"/>
</class>
</hibernate-mapping>
```

EmployeeDAO.java

```java
package com.nareshit.dao;

import org.hibernate.Session;

import com.nareshit.pojo.Employee;
import com.nareshit.utility.HibernateUtility;
public class EmployeeDAO{
        public void createEmployee(Employee emp){
                Session session=HibernateUtility.getSession();
                session.save(emp);
                session.beginTransaction().commit();
        }
        public void getEmployeeDetails(int empNo){
                Session session=HibernateUtility.getSession();
```

```java
        Employee employee=(Employee)session.get(Employee.class,empNo);
    System.out.println("Employee Details");
    System.out.println("================");
    System.out.println("Emp No :"+employee.getEmpNo());
    System.out.println("Emp Name :"+employee.getName());
    System.out.println("Salary :"+employee.getSalary());
    System.out.println("DeptNo :"+employee.getDept().getDeptNo());
    System.out.println("DeptName :"+employee.getDept().getDeptName());
    System.out.println("DeptLocation :"+employee.getDept().getLocation());

    }
}
```

DepartmentDAO.java

```java
package com.nareshit.dao;
import org.hibernate.Session;

import com.nareshit.pojo.Department;
import com.nareshit.utility.HibernateUtility;
public class DepartmentDAO {
public void createDepartment(Department dept){
Session session=HibernateUtility.getSession();
session.save(dept);
session.beginTransaction().commit();
}
}
```

Test.java
```java
package com.nareshit.client;
import java.util.List;
import com.nareshit.dao.DepartmentDAO;
import com.nareshit.dao.EmployeeDAO;
import com.nareshit.pojo.Department;
import com.nareshit.pojo.Employee;
public class Test{
public static void main(String[] args){
        EmployeeDAO empDao=new EmployeeDAO();
        DepartmentDAO deptDao=new  DepartmentDAO();
        System.out.println("Testing of createDepartment(-)");
        Department salesDept=new Department(12,"Sales","hyd");
        deptDao.createDepartment(salesDept);
        Department adminDept=new Department(13,"Admin","hyd");
        deptDao.createDepartment(adminDept);
        System.out.println("Testing of createEmployee(-)");
        Employee emp1=new Employee("sathish",15000.0);
        emp1.setDept(salesDept);
        empDao.createEmployee(emp1);
        Employee emp2=new Employee("ramesh",16000.0);
        emp2.setDept(salesDept);
        empDao.createEmployee(emp2);
```

```java
Employee emp3=new Employee("raju",14000.0);
    emp3.setDept(adminDept);
    empDao.createEmployee(emp3);
    Employee emp4=new Employee("ramu",14500.0);
    emp4.setDept(adminDept);
    empDao.createEmployee(emp4);
    Employee emp5=new Employee("pavan",15500.0);
    emp5.setDept(salesDept);
    empDao.createEmployee(emp5);
    System.out.println("Testing of getEmployeeDetails(-)");
    empDao.getEmployeeDetails(101);
}
}
```

Example 2: Many-to-One AnnnotationMappingExample

- ▲ 🗁 ManytoOne Annotation Mapping Example
  - ▲ 📁 src
    - ▲ ⊞ com.nareshit.client
      - ▷ 🗋 Test.java
    - ▲ 🗄 com.nareshit.config
      - ⌧ hibernate.cfg.xml
    - ▲ ⊞ com.nareshit.dao
      - ▷ 🗋 DepartmentDAO.java
      - ▷ 🗋 EmployeeDAO.java
    - ▲ ⊞ com.nareshit.pojo
      - ▷ 🗋 Department.java
      - ▷ 🗋 Employee.java
    - ▲ 🗁 com.nareshit.utility
      - ▷ 🗋 HibernateUtility.java

<u>Employee.java</u>

```java
package com.nareshit.pojo;
import javax.persistence.CascadeType;
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;

import org.hibernate.annotations.GenericGenerator;

@Entity
@Table(name="Employee_Details")
public class Employee {
    @Id
    @GeneratedValue(generator="myGenerator")
    @GenericGenerator(name="myGenerator",strategy="increment")
```

```java
        private int empNo;
        @Column(length=12)
        private String name;
        private double salary;
        @ManyToOne(cascade=CascadeType.ALL)
        @JoinColumn(name="dept_no",nullable=false)
        private Department dept;
        public Employee(){
        }
        public Employee(String name,double salary){
        this.name=name;
        this.salary=salary;
        }
        public int getEmpNo() {
                return empNo;
        }
        public void setEmpNo(int empNo) {
                this.empNo = empNo;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public double getSalary() {
                return salary;
        }
        public void setSalary(double salary) {
                this.salary = salary;
        }
        public Department getDept() {
                return dept;
        }
        public void setDept(Department dept) {
                this.dept = dept;
        }
}
```

Note :

The @ManyToOne annotation is used to create the many-to-one relationship between the Employee and Department entities. The cascade option is used to cascade the required operations to the associated entity. If the cascade option is set to CascadeType.ALL then all the operations will be cascaded. For instance when you save a Employee object, the associated Department object will also be saved automatically.

Department.java

```java
package com.nareshit.pojo;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name="department")
public class Department {
        @Id
        private int deptNo;
        @Column(length=12)
        private String deptName;
        @Column(length=12,name="location")
        private String loc;
        public Department(int deptNo,String deptName,String loc){
                this.deptName=deptName;
                this.loc=loc;
                this.deptNo=deptNo;
        }
        public Department(){
        }
        public int getDeptNo() {
                return deptNo;
        }
        public void setDeptNo(int deptNo) {
                this.deptNo = deptNo;
        }
        public String getDeptName() {
                return deptName;
        }
        public void setDeptName(String deptName) {
                this.deptName = deptName;
        }
        public String getLoc() {
                return loc;
        }
        public void setLoc(String loc) {
                this.loc = loc;
        }
}
```

Note :- Remaining files are same as the above Many-to-one XmlMapping Example

One-To-Many:-

This type of association relates one entity object to many object's of another entity.

An Example of this type of association is the relationship

Between a department and the collection of Employees.

Employee_Details

empNo (pk)	name	salary	dept_No (fk)
101	sathish	15000.0	12
102	ramesh	16000.0	12
103	raju	14000.0	13
104	ramu	14500.0	13
105	pavan	15500.0	12

Department_Details

deptNo (pk)	deptName	location
12	Sales	hyd
13	Admin	hyd

One-to-Many(pk-->fk)
Many-to-one (fk-->pk)

Employee_Details and Department_Details table contains One-to-many relationship. Each Department can be associated with multiple Employees and each Employee can have only one Department.

One-To-Many Relationship: A relationship in which each record in one table is linked to multiple records in another table

One-to-Many relation can be developed in four ways

1. Set

2. List

3. Bag

4. Array

```
▲ 🗁 OneToManyAndManyToOneXMLMapping
  ▲ 🌐 src
    ▲ ⊞ com.nareshit.client
      ▷ 🗋 Test.java
    ▲ 🗈 com.nareshit.config
        🗋 hibernate.cfg.xml
    ▲ 🗈 com.nareshit.mapping
        🗋 Department.hbm.xml
        🗋 Employee.hbm.xml
    ▲ ⊞ com.nareshit.pojo
      ▷ 🗋 Department.java
      ▷ 🗋 Employee.java
    ▲ ⊞ com.nareshit.util.dao
      ▷ 🗋 DepartmentDAO.java
      ▷ 🗋 EmployeeDAO.java
    ▲ ⊞ com.nareshit.utility
      ▷ 🗋 HibernateUtility.java
```

Employee.java

package com.nareshit.pojo;
public class Employee {
private int empNo;

```java
private String name;
private double salary;
private Department dept;
public Employee(){
}
public Employee(String name,double salary){
this.name=name;
this.salary=salary;
}
public int getEmpNo() {
        return empNo;
}
public void setEmpNo(int empNo) {
        this.empNo = empNo;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public double getSalary() {
        return salary;
}
public void setSalary(double salary) {
        this.salary = salary;
}
public Department getDept() {
        return dept;
}
public void setDept(Department dept) {
        this.dept = dept;
}

}
```

Department.java

```java
package com.nareshit.pojo;

import java.util.List;
import java.util.Set;

public class Department {
        private int deptNo;
        private String deptName,loc;
        private Set<Employee> employees;
        public Department(int deptNo,String deptName,String loc){
                this.deptName=deptName;
                this.loc=loc;
                this.deptNo=deptNo;
```

```java
        }
        public Department(){
        }
        public int getDeptNo() {
                return deptNo;
        }
        public void setDeptNo(int deptNo) {
                this.deptNo = deptNo;
        }
        public String getDeptName() {
                return deptName;
        }
        public void setDeptName(String deptName) {
                this.deptName = deptName;
        }
        public String getLoc() {
                return loc;
        }
        public void setLoc(String loc) {
                this.loc = loc;
        }
        public Set<Employee> getEmployees() {
                return employees;
        }
        public void setEmployees(Set<Employee> employees) {
                this.employees = employees;
        }
}
```

Employee.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Employee" table="Employee_Details">
<id name="empNo" column="empNo">
<generator class="increment"/>
</id>
<property name="name" length="12"/>
<property name="salary" />
<many-to-one name="dept" class="com.nareshit.pojo.Department"
column="dept_no"  not-null="true"/>
</class>
</hibernate-mapping>
```

Department.hbm.xml

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

```xml
<hibernate-mapping>
     <class name="com.nareshit.pojo.Department" table="Department_details">
          <id name="deptNo">
          </id>
          <property name="deptName" length="12" />
          <property name="loc" column="location" length="12" />
          <set name="employees" table="Employee_Details"
          inverse="true" lazy="true" fetch="select">
     <key>
       <column name="dept_no" not-null="true" />
     </key>
                    <one-to-many class="com.nareshit.pojo.Employee" />
          </set>
     </class>
</hibernate-mapping>
```

EmployeeDAO.java
```java
package com.nareshit.util.dao;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import com.nareshit.pojo.Employee;
import com.nareshit.utility.HibernateUtility;
public class EmployeeDAO{
       public void createEmployee(Employee emp){
               Session session=HibernateUtility.getSession();
               session.save(emp);
               session.beginTransaction().commit();
       }
       public Employee getEmployeesById(int empNo){
               Employee emp=null;
               Session session=HibernateUtility.getSession();
               emp=(Employee)session.get(Employee.class,empNo);
               return emp;
       }
       public List<Employee> getEmployeesByName(String name){
               List<Employee> list=null;
               Session session=HibernateUtility.getSession();
               String hql=
         "from com.nareshit.pojo.Employee as e where e.name like ?";
               Query query=session.createQuery(hql);
               query.setParameter(0,"%"+name+"%");
               list=query.list();
               return list;
       }
}
```

DepartmentDAO.java

```java
package com.nareshit.util.dao;
import java.util.List;
```

```java
import org.hibernate.Query;
import org.hibernate.Session;
import com.nareshit.pojo.Department;
import com.nareshit.utility.HibernateUtility;
public class DepartmentDAO {
        public void createDepartment(Department dept){
                Session session=HibernateUtility.getSession();
                session.save(dept);
                session.beginTransaction().commit();
        }
        public Department getEmployeesByDeptName(String deptName){
                Department dept=null;
                Session session=HibernateUtility.getSession();
                String hql=
 "from com.nareshit.pojo.Department as d where d.deptName like ?";
                Query query=session.createQuery(hql);
                query.setParameter(0,deptName);
                List<Department> list=query.list();
                if(list.size()>0){
                        dept= list.get(0);
                }
                return dept;
        }
}
```

hibernate.cfg.xml

```xml
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">
oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">
jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">manager</property>
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="show_sql">true</property>
<property name="hbm2ddl.auto">create</property>
<mapping resource="com/nareshit/mapping/Employee.hbm.xml"/>
<mapping resource="com/nareshit/mapping/Department.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Test.java
```java
package com.nareshit.client;
import java.util.List;
import java.util.Set;
```

```java
import com.nareshit.pojo.Department;
import com.nareshit.pojo.Employee;
import com.nareshit.util.dao.DepartmentDAO;
import com.nareshit.util.dao.EmployeeDAO;
/*
 * @author sathish
 *
 */
public class Test {
        public static void main(String[] args){
                EmployeeDAO empDao=new EmployeeDAO();
                DepartmentDAO deptDao=new  DepartmentDAO();
                System.out.println("create the departmentes ");

                Department salesDept=new Department(12,"Sales","hyd");
                deptDao.createDepartment(salesDept);

                Department adminDept=new Department(13,"Admin","hyd");
                deptDao.createDepartment(adminDept);

                System.out.println("create the employees");

                Employee emp1=new Employee("sathish",15000.0);
                emp1.setDept(salesDept);
                empDao.createEmployee(emp1);

                Employee emp2=new Employee("ramesh",16000.0);
                emp2.setDept(salesDept);
                empDao.createEmployee(emp2);

                Employee emp3=new Employee("raju",14000.0);
                emp3.setDept(adminDept);
                empDao.createEmployee(emp3);

                Employee emp4=new Employee("ramu",14500.0);
                emp4.setDept(adminDept);
                empDao.createEmployee(emp4);

                Employee emp5=new Employee("pavan",15500.0);
                emp5.setDept(salesDept);
                empDao.createEmployee(emp5);

System.out.println("search Employee's By Id");
int empId=2;
Employee emp=empDao.getEmployeesById(empId);
//check whether the emp is null (OR) not
System.out.println("emp No : "+empId+"  Details");
        System.out.println("Name :"+emp.getName());
        System.out.println("Salary :"+emp.getSalary());
  System.out.println("dept No :"+emp.getDept().getDeptNo());
```

```
        System.out.println("dept Name :"+emp.getDept().getDeptName());
System.out.println("Search Employee's By Name : ");
String name="aj";
List<Employee> list=empDao.getEmployeesByName(name);
        for(Employee empl:list){
                System.out.println("emp No : "+empl.getEmpNo()+" Details ");
                System.out.println("Name :"+empl.getName());
                System.out.println("Salary :"+empl.getSalary());

                System.out.println("dept No :"+empl.getDept().getDeptNo());
                System.out.println("dept Name :"+empl.getDept().getDeptName());

        }

System.out.println("Searching the Employee's By Dept Name :");
                String deptName="Sales";
Department dept1=deptDao.getEmployeesByDeptName(deptName);
System.out.println("Dept Name :"+dept1.getDeptName()+" : deptNo : "+dept1.getDeptNo());
Set<Employee> set= dept1.getEmployees();
    for(Employee e:set){
                System.out.println("emp No : "+e.getEmpNo()+" Details ");
                System.out.println("Name :"+e.getName());
                System.out.println("Salary :"+e.getSalary());

        }
        }
}
```

One-to-Many <bag> example

A <bag> is an unordered collection, which can contain duplicated elements. That means if you persist a bag with some order of elements, you cannot expect the same order retains when the collection is retrieved. There is not a "bag" concept in Java collections framework, so we just use a java.util.List correspond to a <bag>.

To implement Bag in our one-to-many mapping example, we will do following changes:

Update Department Model Class

Department.java

Update *Department.hbm.xml mapping file :-*

```
Package com.nareshit.pojo;
import java.util.ArrayList;
import java.util.List;
 public class Department {
   private int deptNo;
    private String deptName,loc;
    private List<Employee> employees;
// Getter and Setter methods
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
     "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
 <hibernate-mapping >
<class name=" com.nareshit.pojo.Department" table="Department_Details">
 <id name="deptNo"  column="deptNo" >  </id>
  <property name="deptName" column="DeptName"/>
<bag name="employees" table="employee_details"  inverse="true" lazy="true" fetch="select">
      <key><column name="dept_no" not-null="true" />
      </key>
      <one-to-many class="com.nareshit.pojo.Employee" />
    </bag>
   </class>
</hibernate-mapping>
```

Execute <bag> example

Execute the same Test.java file that we created in above example.

One-to-Many <list> example

A <list> is an indexed collection where the index will also be persisted. That means we can retain the order of the list when it is retrieved. It differs from <bag> for it persists the element Index while a <bag> does not. The corresponding type of a <list> in Java is java.util.List.

To implement List in our one-to-many mapping example, we will do following changes:

Update Department Model Class

Department.java

```java
Package com.nareshit.pojo;
import java.util.ArrayList;
import java.util.List;
public class Department {
  private int deptNo;
   private String deptName,loc;
   private List<Employee> employees;
// Getter and Setter methods
}
```

Update **Department.hbm.xml mapping file :-**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
 <hibernate-mapping >
<class name=" com.nareshit.pojo.Department" table="Department_Details">
 <id name="deptNo"  column="deptNo" >   </id>
  <property name="deptName" column="DeptName"/>
<list name="employees" table="employee_details"
     inverse="false" cascade="all">
        <key column="dept_no"  />
     <list-index column="idx" />
 <one-to-many class="com.nareshit.pojo.Employee" />
     </list>
   </class>
</hibernate-mapping>
```

In the above hibernate mapping xml file, note that we have added list tag to map a list of employees with department. Also a new index column "idx" is defined which will store the index of records. Note that inverse="false" is specified in the mapping which makes Department as the relationship owner. Thus when department object is saved, it automatically saves the Employees too. This is required so that Department can manage the index values for employees. The dept_no is given as key column.

 Execute <List> example

Execute the same Test.java file that we created in above example.

One to Many relationship using Annotation mapping

Remove Hibernate Mapping (hbm) Files

We are not going to use hibernate mapping files or hbm files as we will map the model using Java 5 Annotations. Delete the files employee.hbm.xml and department.hbm.xml in the above xml mapping example

 Update Employee and Department Model Class as follows

Employee.java

package com.nareshit.pojo;

```java
@Entity
@Table(name="Employee_Details")
public class Employee {
@Id
@GeneratedValue(generator="myGenerator")
@GenericGenerator(name="myGenerator",strategy="increment")
```

```
private int empNo;
@Column(length=12)
private String name;
private double salary;
@ManyToOne
@JoinColumn(name="dept_no")
private Department dept;
```

//getters and setters
}

Department.java

```
package com.nareshit.pojo;
@Entity
@Table(name="department_details")
public class Department {
        @Id
        private int deptNo;
        @Column(length=12)
        private String deptName;
        @Column(length=12,name="location")
        private String loc;
        @OneToMany(mappedBy="dept")
        private Set<Employee> employees;
    //getters and setters
}
```

@OneToMany annotation defines a many-valued association with one-to-many multiplicity. If the collection is defined using generics to specify the element type, the associated target entity type need not be specified; otherwise the target entity class must be specified.

The association may be bidirectional. In a bidirectional relationship, one of the sides (and only one) has to be the owner: the owner is responsible for the association column(s) update. To declare a side as not responsible for the relationship, the attribute mappedBy is used. mappedBy refers to the property name of the association on the owner side.. As you can see, you don't have to (must not) declare the join column since it has already been declared on the owners side.

Inverse :

inverse keyword is responsible for managing the insert / update of the foreign key column.

An inverse keyword has the boolean value "true/false". The Default value of this keyword is 'false'

if the inverse keyword value is false parent class is responsible for saving/updating the child and it's relationship.

if the inverse keyword value is true an associated subclass is responsible for saving/updating itself.

Note :-

An inverse keyword is always used with the one-to-many and many-to-many.

It doesn't work with many-to-one relationship.

Many-to-Many:-

This type of relation relates many objects of an Entity to many object's of another Entity.

For Example:-

A relation ship between doctors and patients one doctor may contains many no.of patients.

one patient may consult(contains) many no.of doctors.

A relation ship between students and courses._ there may be chance many student's registered for one course._(or) a student can register for multiple courses.

This Relation ship is possible only in Bidirectional.In this relation ship parent contains

one (or) more No.of child obj's and child contains one (or) more No.of parent objects.
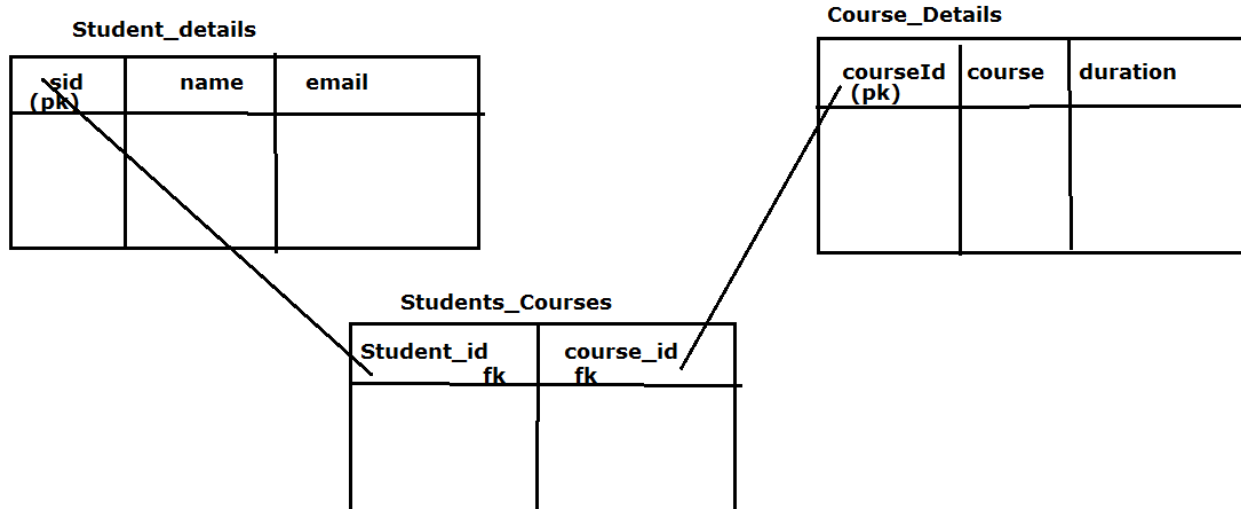
To define a many-to-many relation-ship in the database environment we can use link table(also known as collection table).

In the following example student entity maps to Student_Details table and course entity maps to Course_Details table.we do not have any entity class for the link table (in this case students_courses).

The students_courses table declares a foreign key column named as student_id referring the student_details table,and another fk column named as course_id referring the course_details table.

Many-To-Many Mapping Example (XML Mapping)

- ManytoManyBidirectionalExample
  - src
    - com.nareshit.client
      - Test.java
    - com.nareshit.mapping
      - Course.hbm.xml
      - Student.hbm.xml
    - com.nareshit.pojo
      - Course.java
      - Student.java
    - hibernate.cfg.xml

Student.java
```java
package com.nareshit.pojo;

import java.util.Set;

public class Student {
private int sid;
private String name,email;
private Set<Course> courses;
public Student(){
}
public Student(int sid,String name,String email){
        this.sid=sid;
        this.name=name;
        this.email=email;
}

public int getSid() {
        return sid;
}
public void setSid(int sid) {
        this.sid = sid;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
public String getEmail() {
        return email;
}
public void setEmail(String email) {
        this.email = email;
}
public Set<Course> getCourses() {
        return courses;
```

```
}
public void setCourses(Set<Course> courses) {
        this.courses = courses;
}
}
```

Course.java
```
package com.nareshit.pojo;

import java.util.Set;

public class Course {
private int courseId;
private String courseName,duration;
private Set<Student> students;
public Course(){
}
public Course(int courseId,String courseName,String duration){
this.courseId=courseId;
this.courseName=courseName;
this.duration=duration;

}

public int getCourseId() {
        return courseId;
}
public void setCourseId(int courseId) {
        this.courseId = courseId;
}
public String getCourseName() {
        return courseName;
}
public void setCourseName(String courseName) {
        this.courseName = courseName;
}
public String getDuration() {
        return duration;
}
public void setDuration(String duration) {
        this.duration = duration;
}
public Set<Student> getStudents() {
        return students;
}
public void setStudents(Set<Student> students) {
        this.students = students;
}

}
```

Student.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Student"
 table="student_details">
<id name="sid" >
<generator class="assigned"/>
</id>
<property name="name" length="12"/>
<property name="email" length="20"/>
<set name="courses" table="Students_Courses" cascade="all">
<key column="student_id"/>
<many-to-many class="com.nareshit.pojo.Course"
column="course_id"/>
</set>
</class>
</hibernate-mapping>
```

Course.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Course"
table="course_details">
<id name="courseId" >
<generator class="assigned"/>
</id>
<property name="courseName" length="15"/>
<property name="duration" length="15"/>
<set name="students" table="Students_Courses" cascade="all">
<key column="course_id"/>
<many-to-many class="com.nareshit.pojo.Student"
column="student_id"/>
</set>
</class>
</hibernate-mapping>
```

hibernate.cfg.xml

```xml
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">
```

oracle.jdbc.driver.OracleDriver</property>
<property name=*"connection.url"*>
jdbc:oracle:thin:@localhost:1521:XE</property>
<property name=*"connection.username"*>system</property>
<property name=*"connection.password"*>manager</property>
<property name=*"dialect"*>org.hibernate.dialect.Oracle9Dialect</property>
<property name=*"show_sql"*>true</property>
<property name=*"hbm2ddl.auto"*>update</property>
<mapping resource=*"com/nareshit/mapping/Student.hbm.xml"*/>
<mapping resource=*"com/nareshit/mapping/Course.hbm.xml"*/>
</session-factory>
</hibernate-configuration>

Test.java

```
package com.nareshit.client;

public class Test {

public static void main(String[] args) {

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

        Session session=factory.openSession();

String hql="from com.nareshit.pojo.Course as c where c.courseName=?";

Query query=session.createQuery(hql);

query.setParameter(0,"java");

List<Course> list=query.list();

//check list is empty (OR) not

 Course course=list.get(0);

 System.out.println("Course Name :"+course.getCourseName());

 Set<Student> setOfStudents=course.getStudents();

 for(Student s:setOfStudents){

        System.out.println("Sid :"+s.getSid());

        System.out.println("Name :"+s.getName());

        System.out.println("Email :"+s.getEmail());

     }

   }

}
```

Criteria API

- The Criteria API provides an object-oriented way to retrieve persistent object's.

- This Provides a simplified API to build queries dynamically at runtime.

- Criteria is an alternative way to write the queries without using HQL.

- The Criteria Api allows queries to build at runtime without direct string manipulations.

- Criteria is an API from hibernate and used to write queries in object oriented manner rather than sql(or) HQL.

- Criteria Api is more useful when we have variable no.of conditions in a query.

- We can execute only Select statements by using criteria, we can not execute Update, Delete,statements using crieteria.

- criteria api also include aggregation methods

Steps to work with Criteria API:-

```
step1 : create org.hibernate.Criteria Object
to do this we can call createCriteria(-) method from session

Example :
Criteria criteria=session.createCriteria(Employee.class);

step2 : cretae org.hibernate.criterion.Criterion object per each condition
of the query and to Criteria Object.

to do this we can call Restrictions class utility methods (OR) Property
class methods

Example : Criterion nameCriterion=Restrictions.like("ename","sathish");

        criteria.add(nameCriterion);

step 3:   Execute org.hibernate.Criteria Object
by using list() of criteria object

 List  list=criteria.list();
```

- Criteria is an interface present in org.hibernate package used to represent the query in object oriented manner.It contains methods to add Criterion object's,Order Objects,ProjectionObject's,Pagination methods.....etc.CriteriaImpl is an implementation class of Criteria

To get Criteria Object we can the following methods from session

public Criteria createCriteria(Class entityClass)

public Criteria createCriteria(String entityClassName)

- Restrictions is an utility class present in org.hibernate.criterion package which has methods to define conditions.

- Restrictions class methods returns different XxXExpression classes which implementes Criterion interface .So Restrictions is a static factory for Criterion instances.

- Criterion is an interface present in org.hibernate.criterion package which is used to represent one condition in object oriented manner.

- By calling methods on Restrictions we will get Criterion Object.

RestrictionClass important methods

Expression class	Restrictions method	SQL operator/function
SimpleExpression	eq(prop, val)	=
SimpleExpression	ne(prop, val)	!=, <>
SimpleExpression	like(prop, val) [case-sensitive]	LIKE
SimpleExpression	gt(prop, val)	>
SimpleExpression	lt (prop, val)	<
SimpleExpression	ge(prop, val)	>=
SimpleExpression	le(prop, val)	<=
IdentifierExpression	idEq(val)	<Id-column> =
ILikeExpression	ilike(prop, val) [case-in-sensitive]	LIKE
BetweenExptession	between(prop, val, val)	BETWEEN <lowerLmit> AND <upperLimit>
InExpression	in(prop, val[])	IN(val1, val2....)
NullExpression	isNull(prop)	<column> IS NULL
NullExpression	isNotNull(prop)	<column> IS NOT NULL
LogicalExpression	and (criterion1, criterion2)	<expr1> AND <expr2>
LogicalExpression	or(criterion1, criterion2)	<expr1> OR <expr2>
NotExpression	not(criterioin)	!
EmptyExpression	isEmpty(prop)	EMPTY

Examples :

Example code to read All the Employee records

Session session=sessionFactory.openSession();

Criteria criteria=session.createCriteria("com.nareshit.pojo.Employee");

List<Employee> employees=criteria.list();

for(Employee emp:employees){

System.out.printnln("Name : "+ emp.getName()+" Salary : "+emp.getSalary());

}

Example code to read the Employee records whose name is "ramu"

Session session=sessionFactory.openSession();
Criteria crit = session.createCriteria(Employee.**class**);
Criterion nameCriterion=Restrictions.*eq*("name","ramu");
crit.add(nameCriterion);
List<Employee> results = crit.list();

```
for(Employee emp:results){

System.out.printnln("Name : "+ emp.getName()+" Salary :   "+emp.getSalary());

}
```

Example code to read All the Employee records whose salary is > 8000.0

```
Session session=sessionFactory.openSession();
Criteria crit=session.createCriteria(Employee.class);
crit.add(Restrictions.gt("salary",new Double(8000.0)));
List<Employee> results = crit.list();

for(Employee emp:results){

System.out.printnln("Emp No : "+emp.getEmpNo()+" Name : "+ emp.getName()+"  Salary :
"+emp.getSalary());

}
```

Example code to read   All the Employee records whose name is starts with "ra" and salary is > 8000.0

```
Session session=sessionFactory.openSession();
Criteria crit=session.createCriteria(Employee.class);

Criterion nameCriterion= Restrictions.like("name","ra", MatchMode.START);

Criterion salaryCriterion= Restrictions.gt("salary",new Double(8000.0));
Criterion condition=Restrictions.and(nameCriterion,salaryCriterion);
crit.add(condition);
List<Employee> results = crit.list();

for(Employee emp:results){

System.out.printnln("Name : "+ emp.getName()+" Salary :   "+emp.getSalary());

}
```

org.hibernate.criterion.Property class :

- org.hibernate.criterion.Property is a class available in hibernate API

- Property is used to create Criterion objects(means creating conditions of the query)

- Property is just like Restrictions class,both are meant for creating Criterion objects

- You can create a Property by calling Property.forName("property-name");

public static Property forName(String propertyName)

Example :

Property propertyObj=Property.forName("name");

Property class methods:

- asc() : Order - Property
- avg() : AggregateProjection - Property
- between(Object min, Object max) : Criterion - Property
- count() : CountProjection - Property
- desc() : Order - Property
- eq(DetachedCriteria subselect) : Criterion - Property
- eq(Object value) : SimpleExpression - Property
- eqAll(DetachedCriteria subselect) : Criterion - Property
- eqOrIsNull(Object value) : Criterion - Property
- eqProperty(Property other) : PropertyExpression - Property
- eqProperty(String other) : PropertyExpression - Property
- equals(Object obj) : boolean - Object
- ge(DetachedCriteria subselect) : Criterion - Property
- ge(Object value) : SimpleExpression - Property
- geAll(DetachedCriteria subselect) : Criterion - Property
- geProperty(Property other) : PropertyExpression - Property
- geProperty(String other) : PropertyExpression - Property
- geSome(DetachedCriteria subselect) : Criterion - Property
- getAliases() : String[] - SimpleProjection
- getClass() : Class<?> - Object

Example Code:-

```
Criteria crit=session.createCriteria(Employee.class);
Property nameProperty=Property.forName("name");
Criterion nameCriterion=nameProperty.like("sathish");

Property salaryProperty=Property.forName("salary");
Criterion salaryCriterion=salaryProperty.gt(18000.0);
Criterion criterion=Restrictions.and(namecriterion,salaryCriterion);
crit.add(criterion);
List<Employee> results=criteria.list();
for(Employee emp:results){

System.out.println("Name : "+ emp.getName()+"  Salary  :   "+emp.getSalary());

}
```

Projections :-

If we want to select particular properties (columns) (OR) to work with aggregate functions we can use Projections. we have done this in HQL also, but there we don't need any API support .

As part of HQL query it self we can specify required Properties(column) names.

But in criteria we have seperate API to speicfy required properties.

Steps to work with Projections

Step1 :-

Create org.hibernate.criterion. Projection Object's

Example :

Projection nameProjection=Projections.property("name");

Projection salaryProjection =Projections.property("salary");

Projection is an interface ,which represents one selected property (OR)selected aggregate functions.

Projections is a utilty class which contains methods to create Projection Objects. These methods returns different xxxProjection Objects

Step 2 :-

Create org.hibernate.criterion.ProjectionList Object

Example :

ProjectionList plist=Projections.projectionList();

ProjectionList is a class,which is used to hold multiple Projection Objects

Step 3 :-

add Projection object's to ProjectionList

Example :

plist.add(nameProjection);

plist.add(salaryProjection);

Step 4:-

set projectionList object into criteria Object

Example :

criteria.setProjection(plist);

Examples on Projections :-

Example code to read the Employees names

SQL Query : select name from employee;

Criteria Code :
Session session=sessionFactory.openSession();

Criteria crit = session.createCriteria(Employee.**class**);

Projection nameProjection=Projections.property("name");

crit.setProjection(nameProjection);
List<String> results=crit.list();

for(String name:results)

{
System.out.println("Employee Name :"+name);

}

Example code to read the Employees names and salaries

SQL Query : select name,salary from employee;

Criteria Code :

Session session=sessionFactory.openSession();

Criteria crit = session.createCriteria(Employee.**class**);

Projection nameProjection=Projections.property("name");
Projection salaryProjection=Projections.property("salary");
ProjectionList plist=Projections.projectionList();
Plist.add(nameProjection);
Plist.add(salaryProjection);
crit.setProjection(plist);
List<Object[]> results=crit.list();

for(Object[]obj:results){
System.out.println(obj[0]);

System.out.println(obj[1]);

}

Note:-if only one selected Property is there then we can set **the** Projection Object directly to Criteria,But if we have multiple selected Properties,Then we should go for ProjectionList.

Example code to read the Employees names and salaries whose salary is greater than >8000.0

SQL Query : select name,salary from employee>8000.0;

Criteria Code :

```
Session session=sessionFactory.openSession();
Criteria crit = session.createCriteria(Employee.class);
Projection nameProjection=Projections.property("name");
Projection salaryProjection=Projections.property("salary");
ProjectionList plist=Projections.projectionList();
Plist.add(nameProjection);
Plist.add(salaryProjection);
crit.setProjection(plist);
Criterion salaryCriterion=Restrictions.gt("salary",8000.0);
Crit.add(salaryCriterion);
List<Object[]> results=crit.list();

for(Object[]obj:results){
System.out.println(obj[0]);

System.out.println(obj[1]);

}
```

Pagination methods of Criteria:-

While loading objects from the database instead of loading all objects at a time it is possible to load a particular no.of object's from a given point using methods criteria methods in criteria,we have the following two methods to get Pagentaion behaviour.

public Criteria setFirstResult(int firstResult) : specifies the index of the first row
to retrive from.if not set,rows will be retrived begining from row 0(zero).
public Criteria setMaxResult(int totalResult) : this method is used to specifies the total number of records to be retreived.

public Criteria addOrder(Order o) : used to specifies ordering.

Example code to get the Employee records in ascending order on the basis of salary

```
Crietria crit=session.createCriteria(Employee.class);
crit.addOrder(Order.asc("salary"));
List<Employee> list=c.list();
```
for(Employee emp:list){

System.out.println("name : "+emp.getName()+" salary :"+emp.getSalary());

}

Example code to get the Employee records in decending order on the basis of salary

```
Crietria crit=session.createCriteria(Employee.class);
crit.addOrder(Order.desc("salary"));
List<Employee> list=c.list();
```
for(Employee emp:list){

System.out.println("name : "+emp.getName()+" salary :"+emp.getSalary());

}

Aggregate Functions():

Projections class serves the factory methods which can be used as the aggregate functions as follows.

Example 1 code to get rowCount

SQL : select count(*) from Employee

Criteria Code :

```
List<?> list=

session.createCriteria(Employee.class).setProjection(Projections.rowCount()).list();


System.out.println("Total No.of employees "+ list.get(0));

(OR)
Long rowCount=
(Long)session.createCriteria(Employee.class).setProjection(Projections.rowCount())

                .uniqueResult();
System.out.println("Total No.of employees "+rowCount);
```

Example code to get maxSalary

SQL : select max(salary) from Employee

Criteria Code :

List<?> list= session.createCriteria(Employee.class).setProjection(Projections.max("salary")).list();
System.out.println("Maximum salary is : "+ list.get(0));

Example   code to get minSalary

SQL : select min(salary) from Employee

Criteria Code :

List<?> list= session.createCriteria(Employee.class).setProjection(Projections.min("salary")).list();
System.out.println("Minimum salary is : "+ list.get(0));

Example   code with Multiple functions

SQL : select max(salary),min(salary),avg(salary) from Employee

Criteria Code :

List<?> list= session.createCriteria(Employee.class)

      .setProjection(Projections.max("salary"))

      .setProjection(Projections.min("salary"))

      .setProjection(Projections.avg("salary"))

      .list();

(OR)

Property salaryProperty=Property.forName("salary");

List<?> list=session.createCriteria(Account.class)

      .setProjection(Projections.projectionList()

        .add(salaryProperty.max())

        .add(salaryProperty.min())

```
                                .add(salaryProperty.avg()) )

                        .list();
```

Example code for GroupBy Name

SQL : select count(empno),name from employee Group By Name

<u>Criteria Code :</u>
```
List<?>  list=session.createCriteria(Employee.class)

                .setProjection( Projections.projectionList()

                .add(Projections.count("empno"))

                .add(Projections.groupProperty("name")))

                .list();
```

<u>Example code for Between,AND and OrderBy :-</u>

SQL : select empno,name,salary from employee

                where (salary BETWEEN 3000 AND 6000)

                AND(name like '%sathish%'

                    OR

                    name like '%bandi%'

                    OR

                    name like '%java%'

)

                ORDER By name ASC;

<u>Criteria Code :</u>

```
List<Object[] >  list=session.createCriteria(Employee.class)

                .setProjection(Projections.projectionList()

                .add(Projections.property("empno"))

                .add(Projections.property("name")))
```

```
                .add(Restrictions.between("salary",3000.0,6000.0))

                .add(Restrictions.disjunction()

                    .add(Restrictions.like("name","sathish",MatchMode.ANYWHERE))


                .add(Restrictions.like("name","bandi",MatchMode.ANYWHERE))

                .add(Restrictions.like("name","java",MatchMode.ANYWHERE)))

                .addOrder(Order.asc("name"))

                .list();
```

setResultTransformer

public Criteria
setResultTransformer(ResultTransformer resultTransformer)
used to Set a strategy for handling the query results. This
determines the "shape" of the query result set.

Example : Mapping Criteria Results to Map

Map<String,Object>
map=(Map<String,Object>)session.createCriteria(Employee.class).
setProjection(Projections.projectionList()

.add(Projections.avg("salary"),"avgSalary")

.add(Projections.max("salary"),"maxSalary")

.add(Projections.min("salary"),"minSalary")

.setResultTransformer(Transformers.ALIAS_TO_ENTITY_MAP)

.uniqueResult();

NativeSQL

Native SQL is another technique of performing bulk operations on the data using hibernate.

- IF certain persistence operations are not possible with HQL queries, HQL queries are complex to write Then we can go for NativeSQL.
- By using Native SQL, we can perform both select, non-select operations on the data.
- .Native SQL queries are Database software dependent SQL queries.
- Native SQL queries based persistence logic is Database software dependent.
- Native SQL Allows us to call PL/SQL procedures and functions.
- Native SQL Allows us to place both Named and Positional parameters.

Advantages and Disadvantages of Native SQL :

- We can use the database specific keywords (commands), to get the data from the database
- While migrating a JDBC program into hibernate, the task becomes very simple because JDBC uses direct SQL commands and hibernate also supports the same commands by using this Native SQL
- The main draw back of Native SQL is, it makes the hibernate application as database dependent.

If we want to execute Native SQL Queries on the database then, we need to construct an object of SQLQuery, actually this SQLQuery is an interface extended from Query and it is given in " org.hibernate package "

In order to get an object of SQLQuery, we need to calla method createSQLQuery from session interface.**SQL query object represents native SQL queries.**

public SQLQuery createSQLQuery(String sql)

Example 1:

Session session = sessionFactory.open Session();
Transaction tx=session.beginTransaction();
String sql="insert into employee values(?,?,?)";

SQLQuery sqlQuery=session.createSQLQuery(sql);

sqlQuery.setParameter(0,1001);

sql.Query.setParameter(1,"ramu");

sqlQuery.setParameter(2,6000.0);

int count=sqlQuery.executeUpdate();

tx.commit();

Example 2 :

Example code to read all Employess From Employee Table

Session session = sessionFactory.open Session();
 // Get All Employees
SQLQuery query = session.createSQLQuery("select * from Employee");
List<Object[]> results = query.list();

Example 3:

Example code to read all Employess From Employee Table

Session session = sessionFactory.open Session();
 // Get All Employees
SQLQuery query = session.createSQLQuery("select * from Employee");
query.addEntity(Employee.class);
List<Employee> results= query.list();

Note:-

We can use list() method to execute NativeSQL select queries

 We can't use iterate() to execute native SQL select queries if we are using

java.lang,UnSupportedException will be raised.

There are two types of Native SQL Select queries.

    i) Entity Queries :

     selects all the column values of Database table

     (Results must be mapped with HB POJO class)

     To map the results into Entity Types we can use the following method

     public SQLQuery addEntity(String persistenceClassName)

     public SQLQuery addEntity(Class clz)

ii)   **Scalar queries :**

Selects specific column values of Database tables (results must be mapped with Hibernate Data types)

To do this we can use addScalar(-) method

public SQLQuery addScalar(String columnalias)

public SQLQuery addScalar(String columnalias,Type types)

Note : when we are calling addScalar(-) method we are passing Column alias name,type of column, If we don't give any alias name to column,by default it will   take column name as alias name.

Example 4:

The following example shows how to get entity objects from a native sql query using by addEntity().

```
Session session = sessionFactory.open Session();
 // Get All Employees
SQLQuery query = session.createSQLQuery("select * from Employee");
query.addEntity(Employee.class);
List<Employee> results= query.list();
```

Example 5:

The following example shows how to get entity objects from a native sql query using by addEntity().

Employee Table contains three columns(empno,empname,salary)

```
Session session = sessionFactory.open Session();
 SQLQuery query = session.createSQLQuery("select empno,empname,salary  from Employee");
query.addEntity(Employee.class);
List<Employee> results= query.list();
```

Example 6:

The following Example shows how to Map scalar query results with HB datatypes.

Employee Table contains three columns(empno,empname,salary)

Session session = sessionFactory.open Session();
 SQLQuery query = session.createSQLQuery("select empname,salary from Employee");
query.addScalar("empname", StandardBasicTypes.STRING);
query.addScalar("salary",StandardBasicTypes.DOUBLE);
 List<Object[]> results= query.list();

(OR)

Session session = sessionFactory.open Session();
SQLQuery query = session.createSQLQuery("select empname,salary from Employee");
query.addScalar("empname",Hibernate.STRING);
query.addScalar("salary",Hibernate.DOUBLE);
 List<Object[]> results= query.list();

Note :

In older version of Hibernate we are using org.hibernate.Hibernate to specify the types.

 But in the latest versions we are using org.hibernate.type.StandardBasicTypes.

Note :

When we are using addEntity(-) method we should select all colums then only the result

will assigned into Entity object,if we miss any column,Hibernate fails to assign

the results to Entity Objects.

But Some times we required to select only some columns of the table,but those columns data has to get in the object format,if that is the requirement then we can use Transformers object to map the results to our defined format.

Example 7 :

The following Example shows how to Map results into required format by using ResultTransformer Object.

SQLQuery query=session.createSQLQuery("select empname,salary from Employee");

    query.addScalar("empname", StandardBasicTypes.STRING);

    query.addScalar("salary",StandardBasicTypes.DOUBLE);");

    //Map scalar query results with HB datatypes.

    query.setResultTrasformer(Transformers.aliasToBean(Employee.class));

    List<Employee> list=query.list();

Returning non-managed entities

It is possible to apply a ResultTransformer to native SQL queries, allowing it to return non-managed entities.

SQLQuery query = session.createSQLQuery("SELECT empname,salary FROM Employee")
    query.setResultTransformer(Transformers.aliasToBean(EmployeeDTO.class))
      List<EmployeeDTO> list=query.list();

The above query will return a list of EmployeeDTO which has been instantiated and injected the values of empname and salary into its corresponding properties or fields.

Example 8 :

The following Example shows Aggregate function result mapping :

String sql="select MAX(salary) as maxSal from Employee";

    SQLQuery query=session.createSQLQuery(sql);

    query.addScalar("maxSal",Hibernate.DOUBLE);

    Double maxSalary=(Double)query.uniqueResult();

    System.out.println("Max Salary is "+maxSalary);

Example 9 :

The following Example shows Multiple Aggregate functions result mapping :

String sql="select MAX(salary) as maxSal,MIN(salary) as minSal from Employee";

    SQLQuery query=session.createSQLQuery(sql);

    query.addScalar("maxSal",Hibernate.DOUBLE);

    query.addScalar("minSal",Hibernate.DOUBLE);

    Object[] results=(Object[])query.uniqueResult();

    System.out.println("Max Salary is "+result[0]);

     System.out.println("Min Salary is "+result[1]);

Q) Why iterate(-) is not applicable on native SQL queries and why it is applicableon HQL queries?

A) when iterate(-) is used on HQL query the Hibernate software generates multiple select queries based on Single HQL query but this kind of multiple queries generation is not possible while working with native SQL query because of this Hibernate software sends and execute native SQL query in its database software without any modifications and conversions. Due to these reasons iterate(-) is not applicable on native SQL queries.

Native-SQLExample

Department.java

```
package com.nareshit.pojo;
import java.io.Serializable;
import java.util.*;
public class Department implements Serializable{
private int deptno;
private String deptname,location;
Set<Employee> employees;
//required setters and getters
```

Employee.java

```java
package com.nareshit.pojo;
public class Employee {
private int empno;
private String name;
private double salary;
private Department department;
public Employee() {
}
public Employee(String name,double salary) {
        this.name=name;
        this.salary=salary;
}
//required setters and getters
}
```

HibernateUtility.java

```java
package com.nareshit.utility;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
public class HibernateUtility {
private static SessionFactory factory;
static{
Configuration cfg=new Configuration();
cfg.configure("hibernate.cfg.xml");
factory=cfg.buildSessionFactory();
}
public static Session getSession(){
        Session session=null;
        if(factory!=null){
         session=factory.openSession();
        }
        return session;
}
}
```

Department.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Department"
table="DepartmentDetails">
<id name="deptno" length="12">
</id>
<property name="deptname" length="15"/>
<property name="location" length="15"/>
<set name="Employees"  cascade="all"
```

```
table="employeedetails">
<key column="dept_no" />
<one-to-many  class="com.nareshit.pojo.Employee" />
</set>
</class>
 <sql-query name="getEmployeesByDeptNo">
select *from departmentdetails where deptno=?  </sql-query>
<hibernate-maping>
```

Employee.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Employee"  table="emplyeeDetails">
<id name="empno" length="12">
<generator class="increment"/></id>
<property name="name" length="15"/>
<property name="salary" length="7"/>
<many-to-one name="department" cascade="all"
class="com.nareshit.pojo.Department">
<column name="dept_no" />
</many-to-one>
</class>
</hibernate-mapping>
```

hibernate.cfg.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
     "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
    <hibernate-configuration>
  <session-factory>
       <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
       <property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
       <property name="connection.username">system</property>
   <property name="connection.password">manager</property>
  <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="hbm2ddl.auto">update</property>
<property name="show_sql">true</property>
<mapping resource="Employee.hbm.xml"/>
 <mapping resource="Department.hbm.xml"/>
</session-factory>
<hibernate-comfiguration>
```

Test.java

```
package com.nareshit.client;
```

```java
import java.util*;
import org.hibernate.*;
import org.hibernate.transform.Transformers;
import org.hibernate.type.StandardBasicTypes;
import com.nareshit.pojo.*
import com.nareshit.utility.HibernateUtility;
public class Test {
public static void main(String[] args){
        System.out.println("Testing of getEmployeesByName(-)");
        getEmployeesByName("ajay");
        System.out.println("=========");
        System.out.println("Testing of getEmployeesById(-)");
        getEmployeeById(2);
        System.out.println("=========");
        System.out.println("Testing of getEmployeesBySal(-)");
        getEmployeeDetailsBySal();
        System.out.println("=========");
        System.out.println("Testing of getEmployeesByDept(13)");
        getEmployeesByDept(13);
        System.out.println("=========");
        }
public static void getEmployeesByName(String name){
Session session=HibernateUtility.getSession();
String sql="select * from EmployeeDetails where name=?";
SQLQuery query=session.createSQLQuery(sql);
query.setParameter(0,name);
query.addEntity(Employee.class);
//addEntity() is used to map the results with Entity Types
List<Employee> list=query.list();
for(Employee emp:list){
        System.out.println(emp.getEmpno());
        System.out.println(emp.getName());
System.out.println(emp.getSalary());
        Department d=emp.getDepartment();
        System.out.println(d.getDeptname()+" "+d.getDeptno());
}
}//end of getEmployeesByName(-)
public static void getEmployeeById(int empno){
Session session=HibernateUtility.getSession();
String sql="select e.empno as empno, e.name as empName ,e.salary as sal, e.dept_no as deptno,
d.deptname as deptname, d.location as location from employeeDetails e join departmentdetails d
on e.dept_no=d.deptno where empno="+empno;
SQLQuery query=session.createSQLQuery(sql);
    query.addScalar("empName",StandardBasicTypes.STRING);
```

```java
      query.addScalar("sal",StandardBasicTypes.DOUBLE);
      query.addScalar("deptno",StandardBasicTypes.INTEGER);
      query.addScalar("deptname",StandardBasicTypes.STRING);
      query.addScalar("location",StandardBasicTypes.STRING);
      List<Object[]> list=query.list();
        for(Object[] obj:list){
       for(Object obj1:obj){
       System.out.println(obj1);
          }
          }
   }
public static void getEmployeeDetailsBySal(){
Session session=HibernateUtility.getSession();
 String sql="select empno,name from employeedetails where salary>=?";
 SQLQuery query=session.createSQLQuery(sql);
 query.setParameter(0,12000.0);
 query.addScalar("empno",StandardBasicTypes.INTEGER);
 query.addScalar("name",StandardBasicTypes.STRING);
 query.setResultTransformer(Transformers.aliasToBean(Employee.class));
 List<Employee> list=query.list();
for(Employee emp:list){
        System.out.println(emp.getEmpno());
        System.out.println(emp.getName());
        System.out.println(emp.getSalary());
}
}
public static void getEmployeesByDept(int deptno){
Session session=HibernateUtility.getSession();
SQLQuery query=(SQLQuery)session.getNamedQuery("getEmployeesByDeptNo");
query.setParameter(0,deptno);
query.addEntity(Department.class);
List<Department> list=query.list();
Department dept=list.get(0);
System.out.println("department Name : "+dept.getDeptname());
   Set<Employee> set=dept.getEmployees();
   System.out.println("Employees Details ");
  for(Employee emp:set){
        System.out.println("Emp Num :"+emp.getEmpno());
        System.out.println("Emp Name :"+emp.getName());
        System.out.println("--------------");
  }
}
}
```

Project scenario Search Form :

An Operator as Searching the Employee's by entering any keyword

Screen as follows

And Employee and Department Pojo classes with the mapping :

Department.java

```java
@Entity
@Table(name="department_details")
public class Department {
    @Id
    private int deptNo;
    @Column(length=12)
    private String deptName;
    @Column(length=12,name="location")
    private String loc;
    @OneToMany(mappedBy="dept")
    private Set<Employee> employees;
  //required setters and getters
  }
```

Employee.java

```java
@Entity
@Table(name="Employee_Details")
public class Employee {
@Id
@GeneratedValue(generator="myGenerator")
@GenericGenerator(name="myGenerator",
strategy="increment")
private int empNo;
@Column(length=12)
private String name;
private double salary;
@ManyToOne
@JoinColumn(name="dept_no")
private Department dept;
//required setters and getters    }
```

Database tables :

EMPLOYEE_DETAILS					
Column Name	**Data Type**	**Nullable**	**Default**	**Primary Key**	
EMPNO **(pk)**	NUMBER(10,0)	No	-	1	
NAME	VARCHAR2(12)	Yes	-	-	
SALARY	FLOAT	No	-	-	
DEPT_NO **(fk)**	NUMBER(10,0)	Yes	-	-	
DATEOFJOINING	TIMESTAMP(6)	Yes	-	-	
				1 - 5	

DEPARTMENT_DETAILS				
Column Name	**Data Type**	**Nullable**	**Default**	**Primary Key**
DEPTNO **(pk)**	NUMBER(10,0)	No	-	1
DEPTNAME	VARCHAR2(12)	Yes	-	-
LOCATION	VARCHAR2(12)	Yes	-	-
				1 - 3

And SearchParameter's are storing in a separate object and also searchResults also stored in separate object.

SearchParameter and SearchResults :

SearchParameter.java

```java
public class SearchParameter {
private Integer empNo;
private String empName;
private String deptName;

//required setters and getters

}
```

SearchResults.java

```java
public class SearchResults {
private int empNo;
private String name;
private String deptName;
private double salary;

//required setters and getters

}
```

For the above requirement Code with Native-SQL :

public List<SearchResults> searchEmployees(SearchParameter searchParameters) {

List<SearchResults> searchResults = null;

boolean isFirst=true;

StringBuffer sql=new StringBuffer("SELECT Employee_Details.empNo,"

+ "Employee_Details.name,"

+ "Employee_Details.salary,"

+ "Department_Details.deptName FROM Employee_Details Inner Join Department_Details ON Employee_Details.dept_no=Department_Details.deptno ");

if (searchParameters!= null) {

if (searchParameters.getEmpNo() != null && searchParameters.getEmpNo()> 0) {

if(isFirst){

sql .append(" where empno = " +searchParameters.getEmpNo());

```
        }

        else

        {

sql .append( " AND empno = " +searchParameters.getEmpNo());

        }

        isFirst=false;

        }

        if (searchParameters.getEmpName() != null &&
searchParameters.getEmpName().trim().length()> 0) {

                if(isFirst){

                sql.append(" where name like "+"'%"+searchParameters.getEmpName()+"%'");

                        }

                else

                {

        sql .append( " AND name like "+"'%"+searchParameters.getEmpName()+"%'");

                }

                isFirst=false;

        }


        if (searchParameters.getDeptName() != null &&
searchParameters.getDeptName().trim().length()> 0) {

                if(isFirst){

                sql.append( " where Department_Details.deptName like
"+"'"+searchParameters.getDeptName()+"'");

                }

                else

                {

                        sql .append( " AND Department_Details.deptName like
"+"'"+searchParameters.getDeptName()+"'");
```

```
        }

        isFirst=false;

                }

        }//end of outer if
```

SQLQuery sqlQuery=HibernateUtility.getSession().createSQLQuery(sql.toString());

sqlQuery.addScalar("empNo",StandardBasicTypes.INTEGER);

sqlQuery.addScalar("name",StandardBasicTypes.STRING);

sqlQuery.addScalar("salary",StandardBasicTypes.DOUBLE);

sqlQuery.addScalar("deptName",StandardBasicTypes.STRING);

sqlQuery.setResultTransformer(Transformers.aliasToBean(SearchResults.class));

        searchResults=sqlQuery.list();

        return searchResults;

}

Connection Pooling

In jdbc as a java programmer if we want to obtain a connection

with a database then we have the following two approaches.

1) by using DriverManager class of java.sql

2) by using DataSource interface of javax.sql

Sunmicrosystems given datasource interface inorder to OverCome the problems of DriverManager Class.

Problems with DriverManager.getConnection() approach:

java project (or java application)

component1

....
Connection con1;
con1=DriverManager.
getConnection(url,u,p);
.....
...

physical connection
to the data base

Data base

+

component2

.....
Connection con2;
con2 = DriverManager.
getConnection(url,u,p);
....

physical connection to the data base
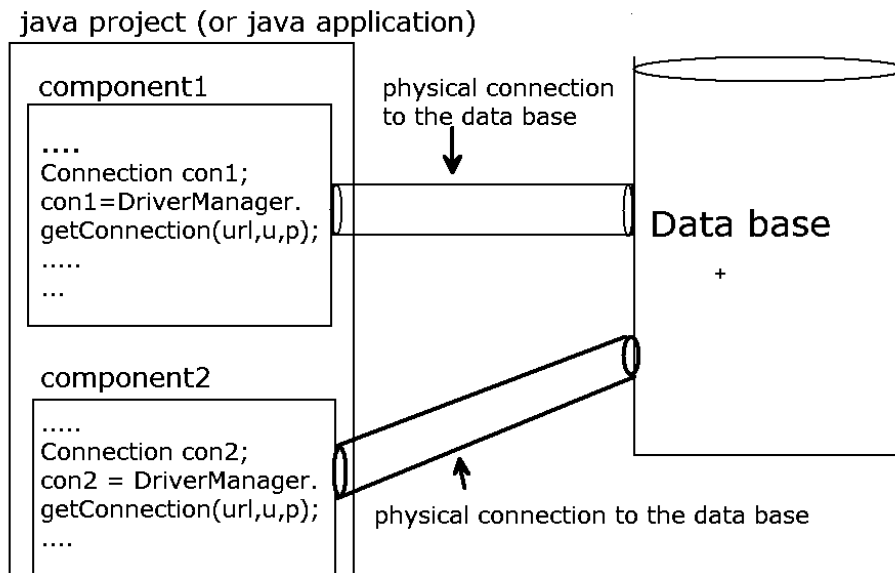
1. **DriverManager class always opens a new Physical connection with the Database**

2. Every time getting the physical connection from the data base takes more time because of the above drawback application performance is reduced.
3. Data base server provider for web-application will charge more why because each new connection request is costly.
4. Even multithreaded environment is also not suggested, we find the problems such as scalability, consistency... etc.

To solve the above problems we must use **connection pooling**.

What is connection pool?

ans: connection pool is a group of logical data base connections associated with a physical data base connection.

with Connection pooing,we have the following two Advantages

1) A connection pool contians a set of Readymade logical Connections(Already open Connections).

So an Application as no need to wait until the Connection is opened.The Application gets

Connection faster.

2) A connection in a Connection pool's are Reusable. because once an application as completed the work then the Connection goes back to the pool. And Same Connection will be reused for next time.

DateSource :

DataSource will act as a meditator between a java application and a Connection pool

DataSource will creates the logical connection to java application

when ever the application closes the logical connection

DataSource will send that connection back to pool.

DataSource is an interface present in javax.sql package .

For DataSource interface different database vendors ,Server Vendors and Framework vendors are

Provided the implementations

Generally we are using Servervendor provided implementations in the realtime project's
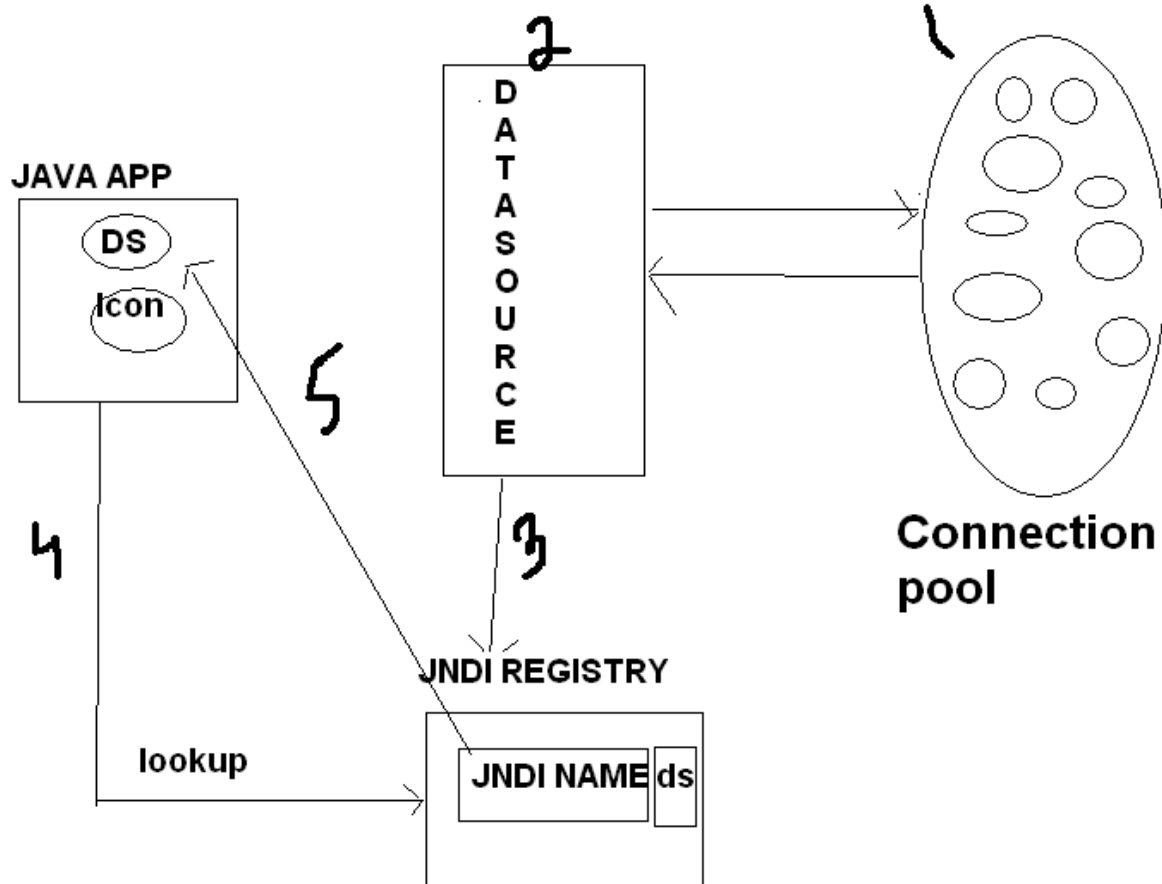
server Adminstartor creates a connection pool with Database.
The Adminstrator creates a mediator Object for the Connection pool called DataSource.
the Administrator binds the DataSource Object into JNDI Registry with some key called
as a JNDI NAME.
the Application Developers communicates with JNDI registry and Obtains ds object from
Registry. the Client Application Connects with Data Source and Obtain the Connection from the
Connection pool

Datasource Object is registred with Naming Service by using naming API lookup() method get
DataSource Object reference.

Example code In Jdbc Program :-

InitialContext ic=new IntialContext();

DataSource dataSource=ic.lookup("registredName");

By calling zero-arg getConnection() method on the DataSource object,java application get a pooled Connection

Connection con=dataSource.getConnection();

By default Every application of Hibernate uses the Hibernate s/w

supplied Built in Jdbc Connection pool which contains set of reademade

jdbc connection objects hibernate SessionFactory obj represents this jdbc connection pool

Note :-

But Don't use hibernate s/w built in connection pooling because it is not good for production

due to it's poor performance.

we can make hb application working with 3 types of jdbc connection pools

        1) Hibernate Software Supplied and 1managed built in jdbc connection pool

        2) Third party vendor s/w managed jdbc connection pool like(c3po and proxool)

        3) web server /app server managed jdbc connection pool

Note :

to guide hb s/w to work with certain jdbc connection pool we need to specify the connection

provider class in hibernate cfg file.

c3po connection pool :

c3p0 is an easy-to-use library used for providing connection pooling capability.

C3P0 is an open source JDBC connection pool distributed along with Hibernate in the `lib` directory. Hibernate will use its `org.hibernate.connection.C3P0ConnectionProvider` for connection pooling .

procedure to work with c3po connection pool in hibernate Application :

step 1:

configure the c3po pool related connection provider class in hibernate connfiguration file

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

org.hibernate.connection.C3P0ConnectionProvider

</property>

step 2 :

configure the following properties in configuration file realated to c3po pool

<property name="hibernate.c3p0.min_size">10</property>

<property name="hibernate.c3p0.max_size">30</property>

<property name="hibernate.c3p0.timeout">100</property>

<property name="hibernate.c3p0.max_statements">50</property>

<property name="hibernate.c3p0.idle_test_period">1000</property>

These are the basic things you need to know about c3p0 configuration properties:

initialPoolSize C3P0 default: 3

minPoolSize Must be set in hibernate.cfg.xml (or hibernate.properties), Hibernate default: 1

maxPoolSize Must be set in hibernate.cfg.xml (or hibernate.properties), Hibernate default: 100

i**dleTestPeriod** Must be set in hibernate.cfg.xml (or hibernate.properties), Hibernate default: 0

If this is a number greater than 0, c3p0 will test all idle, pooled but unchecked-out connections, every this number of seconds.

timeout Must be set in hibernate.cfg.xml (or hibernate.properties), Hibernate default: 0

The seconds a Connection can remain pooled but unused before being discarded. Zero means idle connections never expire.

maxStatements Must be set in hibernate.cfg.xml (or hibernate.properties), Hibernate default: 0

The size of c3p0's PreparedStatement cache. Zero means statement caching is turned off.

propertyCycle Must be set in c3p0.properties, C3P0 default: 300

maximum time in seconds before user configuration constraints are enforced. c3p0 enforces configuration constraints continually, and ignores this parameter. It is included for JDBC3 completeness.

acquireIncrement Must be set in hibernate.cfg.xml (or hibernate.properties), Hibernate default: 1

Determines how many connections at a time c3p0 will try to acquire when the pool is exhausted.

testConnectionOnCheckout Must be set in c3p0.properties, C3P0 default: false

Don't use it, this feature is very expensive. If set to true, an operation will be performed at every connection checkout to verify that the connection is valid. A better choice is to verify connections periodically using c3p0.idleConnectionTestPeriod.

autoCommitOnClose Must be set in c3p0.properties, C3P0 default: false

The JDBC spec is unfortunately silent on what should happen to unresolved, pending transactions on Connection close. C3P0's default policy is to rollback any uncommitted, pending work. (I think this is absolutely, undeniably the right policy, but there is no consensus among JDBC driver vendors.) Setting autoCommitOnClose to true causes uncommitted pending work to be committed, rather than rolled back on Connection close. [Note: Since the spec is absurdly unclear on this question, application authors who wish to avoid bugs and inconsistent behavior should ensure that all transactions are explicitly either committed or rolled-back before close is called.]

Here Sample hibernate cfg file with c3p0 connection pool configuration :-

<hibernate-configuration>

<session-factory>
<property name=*"hibernate.connection.driver_class"*>oracle.jdbc.driver.OracleDriver</property>
<property name=*"hibernate.connection.url"*>jdbc:oracle:thin:@localhost:1521:XE
<property name=*"hibernate.connection.username"*>system</property>
<property name=*"hibernate.connection.password"*>manager</property>
<property name=*"hibernate.connection.dialect"*>org.hibernate.dialect.Oracle9Dialect</property>
<property name=*"hibernate.connection.provider_class"*>org.hibernate.connection.C3P0ConnectionProvider</property>
<property name=*"hibernate.c3p0.max_size"*>30</property>
<property name=*"hibernate.c3po.min_size"*>5</property>
<property name=*"hibernate.c3p0.timeout"*>5000</property>
<property name=*"hibernate.c3p0.acquire_increment"*>2</property>
<mapping resource=*"employee.hbm.xml"*/>
</session-factory>

</hibernate-configuration>

Proxool Connection pooling :

Proxool is another open source JDBC connection pool distributed along with Hibernate in the `lib/` directory. Hibernate uses its
`org.hibernate.service.jdbc.connections.internal.ProxoolConnectionProvider` for connection pooling if you set the hibernate.proxool.* properties. Unlike c3p0, proxool requires some

additional configuration parameters, as described by the Proxool documentation available at
http://proxool.sourceforge.net/configure.html.

Here Sample hibernate cfg file for Proxool Connection pool :-

Step 1:

```
<hibernate-configuration>

<session-factory>
<property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver
</property>
<property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:XE
<property name="hibernate.connection.username">system</property>
<property name="hibernate.connection.password">manager</property>
<property name="hibernate.connection.dialect">org.hibernate.dialect.Oracle9
Dialect</property>
<property
name="hibernate.connection.provider_class">org.hibernate.connection.ProxoolConnectionProvider
</property>
<property name="hibernate.proxool.pool_alias">jdbcpool</property>
<property name="hibernate.proxool.xml">myfile.xml</property>
<mapping resource="employee.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Step2:

myfile.xml

```
<proxool-config>
<proxool>
<alias>jdbcpool</alias>
<driver-url>jdbc:oracle:thin:@localhost:1521:XE </driver-url>
<driver-class> oracle.jdbc.driver.OracleDriver </driver-class>
<driver-properties>
<property name="user" value="system"></property>
<property name="password" value="manager"></property>
</driver-properties>
<minimum-connection-count>10</minimum-connection-count>
<maximum-connection-count>20</maximum-connection-count>
```

</proxool>
</proxool-config>

Apache Dbcp Connection Pool

**In order to integrate this pool with Hibernate you will need the following jars:
commons-dbcp.jar and commons-pool-1.5.4.jar.**

Apache Dbcp Connection Pool **can be downloaded from
http://commons.apache.org/dbcp/**

Here's a sample configuration in hibernate.cfg.xml:

<hibernate-configuration>

 <session-factory>

<property name=*"hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver
</property>*
<property name=*"hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:XE*
<property name=*"hibernate.connection.username">system</property>*
<property name=*"hibernate.connection.password">manager</property>*
<property name=*"hibernate.connection.dialect">org.hibernate.dialect.Oracle9
Dialect</property>*
  <property name="hibernate.dbcp.initialSize">8</property>

 <property name="hibernate.dbcp.maxActive">20</property>

 <property name="hibernate.dbcp.maxIdle">20</property>

 <property name="hibernate.dbcp.minIdle">0</property>

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

</session-factory>

</hibernate-configuration>

Obtaining connections from an application/web server, using JNDI

To use Hibernate inside an application server, configure Hibernate to obtain connections from an application server `javax.sql.Datasource` registered in JNDI, by setting at least one of the following properties:

- hibernate.connection.datasource (required)
- hibernate.jndi.url
- hibernate.jndi.class
- hibernate.connection.username
- hibernate.connection.password

JDBC connections obtained from a JNDI datasource automatically participate in the container-managed transactions of the application server.

Tomcat JNDI DataSource Configuration :

For configuring tomcat container to initialize DataSource, we need to make some changes in tomcat context.xml files.

Context.xml file

```
<Resource name="jdbc/OracleDB "

        type="javax.sql.DataSource"

        driverClassName="oracle.jdbc.driver.OracleDriver "

        url="jdbc:oracle:thin:@localhost:1521:XE"

        username="system"

        password="manager"

        maxActive="30"

        maxIdle="20"

        minIdle="5"

        maxWait="10000"/>
```

By using JNDI Configuration

hibernate.cfg.xml

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

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

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

<hibernate-configuration>

  <session-factory>

  <property name="hibernate.dialect">

    org.hibernate.dialect.Oracle9Dialect</property>

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

    java:comp/env/jdbc/OracleDB</property>

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

  </session-factory>

</hibernate-configuration>

Hibernate Filters :

Filters are used to select some specific data from the database.

A filter in the HB provides environment to pass conditions of the persistent logic from outside the java sources through HB mapping file. A HB filter is a global, named, parameterized filter representing conditions like where clause. Having the ability of enabling or disabling them for a particular Hibernate Session.

Working with filters is nothing but making the conditions of the queries coming to application through hibernate mapping file.

Filters of hibernate are introduced from Hibernate 3.x version.

Steps to work with HB Filters to filter the data through the restrictions:

1)Define filter having parameters in Hibernate mapping file

<hibernate-mapping>

<class>

……

……

</class>

```
<filter-def name="myFilter">
```

```
<filter-param name="mId1″ type=" java.lang.Integer"/>
```

```
<filter-param name="myId2″ type=" java.lang.Integer"/>
```

```
</filter-def>
```

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

2)link/configure filter with Hibernate pojo class through the Hibernate mapping file.

```
<hibernate-mapping>
```

```
<class>
```

……

……

```
<filter name="MyFilter" condition="empId >= :myId1 and empId &lt;= :myId2″></filter>
```

```
</class>
```

```
<filter-def name="myFilter">
```

```
<filter-param name="myId1″ type="java.lang.Integer"/>
```

```
<filter-param name="myId2″ type=" java.lang.Integer"/>
```

```
</filter-def>
```

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

3)Enable filter on Hibernate Session Object and supplies values for filter parameters.

Filter filter = session.enableFilter("myFilter");

filter.setParameter("myId1″,new Integer(1001));

filter.setParameter("myId2″,new Integer(1004));

When to use Hibernate Filters

Suppose a web application that does the reporting for various flights. In future course there is some changes in requirement such that flights are to be shown as per their status (on time, delayed or cancelled).

This can also be done using a WHERE clause within the SQL SELECT query or Hibernate's HQL SELECT query. For a mini application it is fine to do this, but for a huge and complex application it might be a tiresome effort. Although it will be like searching each and every SQL query and making the changes in the existing code. But it has been thoroughly tested.

This can also be done using Hibernate's Criteria API but that also means changing the code at numerous places that is all working fine. Moreover in both the approaches, one needs to be very cautious so that they are not changing existing working SQL queries in inadvertent way.

Filters can be used like database views and data dictionary, but parameterized inside any application. This way they are essential when developers have very little control over DB operations. Here weare going to show you the usage of Hibernate filters to solve this problem. When the end users select the status, our application activates the flight's status for the end user's Hibernate session.

Any SQL query will only return the subset of flights with the user chooses status. Flight status is maintained at two places- Hibernate Session and flight status filter.

Getting started with the example

First define filters in the Hibernate mapping documents, using the <filter-def> XML element.

These filter declaration should contain the name of the filter and the names and types of any filter parameters. Demonstrate filter parameters with the <filter-param> XML element. Filter parameters are same to the named parameters for HQL queries. We need to specify a : (colon) before the parameter condition. Here is the mapping file from the code.

Listing 1: Mapping File

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD "
"http://hibernate.sourceforge.net/hibernate-mapping">
<hibernate-mapping package="data">
<class name="Flight" table="hibernate_filter_demo.flightdb">
<id name="id" type = "int" column="id">
<generator class = "increment"/>
</id>
<property name="fliteNo" type="string" length="10" column="flight_No"/>
<property name="src" type="string" length="15" column="source"/>
<property name="dest" type="string" length="15"column="destination"/>
<property name="info" type="string" length="20" column="info"/>
<strong><filter name="statusFilter" condition=":statusParam=status"/>
</class>
<filter-def name="statusFilter">
<filter-param name="statusParam" type="string"/>
</filter-def></strong>
</hibernate-mapping>
```

Now attach the filters to class or collection mapping elements. We can join a single filter to more than one class or collection. To do this, you add a <filter> XML element to each class or collection. The <filter> XML element has two attributes viz. values and condition. The value references a filter definition while condition is analogous to a WHERE clause in HQL. Let's see the complete coding from the HibernateFilters.zip archive.

Note : Each <filter> XML element must correspond to a <filter-def> element. It may be possible to have more than one filter for each filter definition, and each class may have more than one filter.

Concept is to define all the filter parameters in one place and then refer them in the individual filter conditions.

In the java code, we can programmatically enable or hide the filter. By default the Hibernate Session doesn't have any filters visible on it.

Methods of Session Interface:

- public Filter enableFilter(String NameofFilter)

- public Filter getEnabledFilter(String NameofFilter)

- public void disableFilter(String NameofFilter)

Methods of Filter Interface:

- public Filter setParameter(String NameofFilter, Object value)

- public Filter setParameterList(String NameofFilter, Collection values)

- public Filter setParameterList(String NameofFilter, Object[] values)

setParameter() method is almost used. Be careful and define only the type of java object that we have mentioned in the parameter at the time of defining filter in the mapping file.

The two setParameterList() methods are useful for using IN clauses in your filters. If we want to use BETWEEN clauses, must use two different filter parameters with different names.

At the time of enabling the filter on session-use the name that you have provided in the mapping file for the filter name for the corresponding column in the table. Similarly string name should contain one of the possible values for that particular column.

Listing 2: This condition is set on the filter

```
public class HibernateFilterDemo{
public static void main(String argsp[]){
SessionFactory factory = HibernateUtil.getSessionFactory();
Session session = factory.openSession();
insertData("DL6159", "RFC", "JCK", "dealy 15 min", session);
......
<strong>Filter filter = session.enableFilter("statusFilter");
filter.setParameter("statusParam", "delayed");</strong>
showData(session);
.........
session.close();
}
public static void insertData(String fliteNo, String src,
String dest, String info, Session ses){
session.beginTransaction();
```

```
session.getTransaction().commit();
}
public static void showData(Session session){
session.beginTransaction();
Query query = session.createQuery("from Flight");
.........
session.getTransaction().commit();
}
}
```

Conclusion

Hibernate filters is one of the best optionto DB views, Hibernate Criteria API and SQL where clause.It is an efficient way to separate database concerns from the remaining application code. They help in minimizing the complexity of HQL or SQL queries. Filters can be made visible/on as and when they are required.

Example Program on Filters

<u>User.java</u>

```
public class User{
   private int id;
   private String name;
   private boolean activated;

      public boolean isActivated(){
      return activated;
   }
   public void setActivated(boolean activated){
      this.activated = activated;
   }
   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;
   }
}
```

<u>User.hbm.xml</u>

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

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


<hibernate-mapping>
 <class name="User" table="user_data">
  <id name="id" type="int">
   <generator class="increment"/>
  </id>
<property name="name" type="string" length="32"/>
  <property name="activated" type="boolean"/>
  <filter name="activatedFilter"
  condition=":activatedParam = activated"/>
 </class>
 <filter-def name="activatedFilter">
  <filter-param name="activatedParam" type="boolean"/>
 </filter-def>
</hibernate-mapping>
```

hibernate.cfg.xml

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
        PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">


<hibernate-configuration>
  <session-factory>
    <property
name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
    <property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
    <property name="hibernate.connection.username">system</property>
    <property name="connection.password">manager</property>
    <property name="hibernate.dialect">org.hibernate.dialect.OracleDialect</property>
    <property name="hibernate.autocommit">true</property>
    <property name="show_sql">false</property>
    <mapping resource="User.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

SimpleFilterExample.java

```java
import java.util.Iterator;

import org.hibernate.cfg.Configuration;
import org.hibernate.Filter;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

public class SimpleFilterExample{
    public static void main (String args[]){
        SessionFactory factory = new Configuration().configure().buildSessionFactory();
        Session session = factory.openSession();
        //insert the users
    insertUser("pavan",true,session);
    insertUser("chiru",false,session);
 insertUser("ramu",false,session);
  //Show all users
    System.out.println("===ALL USERS===");
    displayUsers(session);
    //Show activated users
  Filter filter = session.enableFilter("activatedFilter");
  filter.setParameter("activatedParam",new Boolean(true));
    System.out.println("===ACTIVATED USERS===");
    displayUsers(session);
     //Show non-activated users
    filter.setParameter("activatedParam",new Boolean(false));
    System.out.println("===NON-ACTIVATED USERS===");
    displayUsers(session);
      session.close();
    }
    public static void displayUsers(Session session){
     Query query = session.createQuery("from User");
      Iterator results = query.iterate();
      while (results.hasNext()){
         User user = (User) results.next();
        System.out.print(user.getName() + " is ");
        if (user.isActivated()){
           System.out.println("activated.");
        }
        else   {
           System.out.println("not activated.");
        }
      }
     }
    public static void insertUser(String name, boolean activated, Session session){
      Transaction trans = session.beginTransaction();
      User user = new User();
      user.setName(name);
```

```
        user.setActivated(activated);
        session.save(user);
         trans.commit();
      }
}
```

Caching

Cache/buffer is a temprary memory that holds data for temporary period.The Process of Storing data in cache/buffer and using that data accross the muliple request/execution is

called as caching/buffering.

In client server environment the cache/buffer that resides at client side holds

results given by server application and reduces network round trips between client and server accross the multiple same request.

The Buffer of Browser Window reduces n/w round trips b/w Browser window and website/web server. caching in HB resides with HB based client App,holds HB pojo class obj's of selected records and reduces n/w round trips b/w client app and db s/w

To get maximum benifts from caching we need to delete connect of cache/buffer that resides with client app at regular Intervels.

Hibernate supports two levels of cahing.

first level cache-->Resides at Session object.specific to each Session object.it is built in cache.

second level cache-->Resides at SessionFactory obj it is not built in cache ,configure explicitly.

first and second level caches of HB s/w ,maintains the results(records) given by db table in the form of HB pojo class objects.

first level cache is called local cache to session object second level cache is called as global cache.

caching staratagies:-

Read only:-

This Strategy is use full for data that is read frequently but never updated this is by far the simplest and best performing cache strategy.

Read/Write:-

This cache may be appropriate if your data needs to be updated. They carry more overhead than read-only cahces.In non-JTA environment each Transaction should be compilicated when session.close()(or) session.disconnect() is called.

Non-strict read/write:-

This startegy doesnot gurantee that two transactions can not simultaneously modify the same data.

Therefore it may be most appropriate for data that is read often but only occasionally modified.

Transactional:-

This is fully transactional cache that may be used only in JTA enviroment.

There are so many second level caching implementations

EHcache:-

EH cache is lightweight and easy to use in process cache. it supports read only,readwrite,

memory and disk-based caching . It does not supports clustring.

OS cache:-

it is another open source caching solution.

It is part of larger package,Which also provides caching for JSp pages (or) arbitary objects.

it supports read only,readwrite,memory and disk-based caching. it also provides basic support for clustring via either java groups(or) JMS

SwarmCache:-

It is Simple clusted based caching solution based on java groups

it supports read only,non-strict read and write i,e this type cache is suitable for applications that

typically have many more read operations than write operations.

JBoss TreeCache:-

It is a power ful replicated(Synchronous(or) asynchronus) and transactional cache.It supported in JTA Environment

Configuring EhCache

To configure ehcache, you need to do two steps:

configure Hibernate for second level caching specify the second level cache provider

Hibernate 3.3 and above

<property key="hibernate.cache.use_second_level_cache">true</property>

<property name="hibernate.cache.region.factory_class">net.sf.ehcache.hibernate.EhCacheRegionFactory</property>

Hibernate 3.2 and below

<property key="hibernate.cache.use_second_level_cache">true</property>

<property
name="hibernate.cache.region.provider_class">net.sf.ehcache.hibernate.EhCacheProvider</property>

Configuring entity objects

This may done in two ways.

If you are using hbm.xml files then use below configuration:

<class name="com.nareshit.pojp.Employee" table="employee">

  <cache usage="read-only"/>

</class>

2) if you are using annotations, use these annotations:

@Entity

@Cache(usage=CacheConcurrencyStrategy.READ_ONLY,

region="employee ")

public class Employee implements Serializable

{

  //code

}

Configure the cache Provider class name in hibernate cfg file:

```
 <property
name="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</property>
```

We can use below configuration to override the default configuration in ehcache.xml

<cache

  name="com.nareshit.pojo.Employee"

---

maxElementsInMemory="10"

eternal="false"

timeToIdleSeconds="300"

timeToLiveSeconds="600"

overflowToDisk="true"

/>

Please note that in ehcache.xml, **if eternal="true" then we should not write timeToIdealSeconds, timeToLiveSeconds**, hibernate will take care about those values
So if you want to give values manually better use eternal="false" always, so that we can assign values into timeToIdealSeconds, timeToLiveSeconds manually.

timeToIdealSeconds="seconds" means, if the object in the global cache is ideal, means not using by any other class or object then it will be waited for some time we specified and deleted from the global cache if time is exceeds more than timeToIdealSeconds value.

timeToLiveSeconds="seconds" means, the other Session or class using this object or not, i mean may be it is using by other sessions or may not, what ever the situation might be, once it competed the time specified timeToLiveSeconds, then it will be removed from the global cache by hibernate.

The following example showing EhCachingExample :

Product.java

```
package com.nit;
public class Product{
private int productId;
private String proName;
private double price;
//required setters and getters
}
```

ehcache.xml

<ehcache>

<defaultCache maxElementsInMemory="100" eternal="false"

    timeToIdleSeconds="120" timeToLiveSeconds="200" />

 <cache name="com.nit.Product" maxElementsInMemory="5" eternal="false"

    timeToIdleSeconds="5" timeToLiveSeconds="200" />

</ehcache>

hibernate.cfg.xml

```xml
<!DOCTYPE hibernate-configuration PUBLIC

    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>

<property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>

<property name="connection.username">system</property>

<property name="connection.password">manager</property>

<property name="cache.provider_class"> org.hibernate.cache.EhCacheProvider</property>

<property name="dialect">org.hibernate.dialect.OracleDialect</property>

<property name="show_sql">true</property>

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

</session-factory>
```

Product.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.nit.Product" table="products">
 <cache usage="read-only" />
  <id name="productId"   />
 <property name="proName" column="name" length="10"/>
  <property name="price"/>
    </class> </hibernate-mapping>
```

```java
package com.nit;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class Test{
public static void main(String[] args){
Configuration cfg = new Configuration();
cfg.configure("hibernate.cfg.xml");
```

```
SessionFactory factory = cfg.buildSessionFactory();
Session session1 = factory.openSession();
Object o=session1.load(Product.class,new Integer(1003));
Product s=(Product)o;
System.out.println("Loaded object product name is___"+s.getProName());
System.out.println("Object Loaded successfully.....!!");
session1.close();
System.out.println("-----------------------------");
System.out.println("Waiting......");
try{
Thread.sleep(4900);
}
catch (Exception e) {
}
System.out.println("4.0 seconds compelted......!!!!!!!!");
Session session2 = factory.openSession();
Object o2=session2.load(Product.class,new Integer(1003));
Product s2=(Product)o2;
System.out.println("Loaded object product name is___"+s2.getProName());
System.out.println("Object loaded successfully...!!");
session2.close();
try{
Thread.sleep(6000);
}
catch (Exception e) {
}
System.out.println("6 seconds compelted......!!!!!!!!");
Session session3 = factory.openSession();
Object o3=session3.load(Product.class,new Integer(1003));
Product s3=(Product)o3;
System.out.println("Loaded object product name is___"+s3.getProName());
System.out.println("Object loaded  successfully.....!!");
session3.close();
factory.close();
        }}
```

Caching -Example 2:-

SearchStuedentController.java

```
package com.nareshit.controller;
import java.io.*;
import javax.servlet.*;
import com.nareshit.dao.StudentDAO;
import com.nareshit.pojo.Student;
public class SearchStudentController extends GenericServlet{
```

```java
private StudentDAO studentDAO;
public void init(){
studentDAO =new StudentDAO();
}
public void service(ServletRequest request,
ServletResponse response)throws ServletException,IOException{
PrintWriter out = response.getWriter();
        String error_msg=null;
        try{
int sid =Integer.parseInt(request.getParameter("sid"));
         Student s =studentDAO.getStudent(sid);
         if(s==null){
        error_msg="Student Not Found";
         }
         else {
        request.setAttribute("student",s);
RequestDispatcher rd =
 request.getRequestDispatcher("SearchStudentView.jsp");
        rd.forward(request, response);
         }
        }//end of try
        catch(NumberFormatException ne){
        error_msg="Please Enter Valid Student Id ";
        System.out.println(ne);
        }//end of catch
        response.setContentType("text/html");
        RequestDispatcher rd =
        request.getRequestDispatcher("SearchStudentForm.html");
        rd.include(request, response);
        out.println("<hr/><i>");
        out.println(error_msg);
        out.println("</i>");
        }//end of service
```

StudentDAO.java

```java
package com.nareshit.dao;
import org.hibernate.Session;
import com.nareshit.pojo.Student;
import com.nareshit.utility.HibernateUtility;
public class StudentDAO {
public Student getStudent(int sid){
Student std=null;
Session session=HibernateUtility.getSession();
```

```
System.out.println("StudentDAO getStudent");
if(session!=null){
    std=(Student)session.get(Student.class,sid);
}
return std;
}
```

Student.java

```
package com.nareshit.pojo;
public class Student {
private int sid,age;
private String name,email;
//required setters and getters
}
```

ehcache.java

```xml
<ehcache>
 <defaultCache  maxElementsInMemory="100" eternal="false"
    timeToIdleSeconds="120" timeToLiveSeconds="200" />
    <cache name="com.nareshit.pojo.Student"  maxElementsInMemory="5" eternal="false"
    timeToIdleSeconds="100" timeToLiveSeconds="200" />
    </ehcache>
```

hibernate.cfg.xml

```xml
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">manager</property>
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="show_sql">true</property>
<property name="cache.provider_class">
  org.hibernate.cache.EhCacheProvider
</property>
<mapping resource="Student.hbm.xml"/>
</session-factory>
```

Student.hbm.xml

```xml
<!DOCTYPE hibernate-mapping PUBLIC
```

```
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.nareshit.pojo.Student" table="Student">
 <cache usage="read-only" />
<id name="sid"></id>
<property name="name"></property>
<property name="age"></property>
<property name="email"></property>
</class>
```

web.xml

```
<web-app>
<servlet>
<servlet-name>searchStudentServlet</servlet-name>
<servlet-class>com.nareshit.controller.SearchStudentController
</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>searchStudentServlet</servlet-name>
<url-pattern>/searchStudent</url-pattern>
</servlet-mapping>
<welcome-file-list>
<welcome-file>SearchStudentForm.html</welcome-file>
</welcome-file-list>
</web-app>

<!-- SearchStudentForm.html -->
<b>Student Management System</b>
<br/>
<form action="searchStudent"><pre>
Student ID : <input type="text" name="sid"/>
<input type="submit" value="Search"/>
```

SearchStudentView.jsp

```
<%@ page import="com.nareshit.pojo.Student"%>
<%!
Student s=null;
%>
<html>
<head><center><h1>Student Management System</h1>
</center></head><br/><br/><hr/>
<body bgcolor="#FFFFF">
```

```
<%
 s=(Student)request.getAttribute("student");
  %>
<center><TABLE BORDER=1>
<tr><th>SID</th><th>NAME</th>
<th>AGE</th><th>EMAIL</th></tr>
<tr>
<td><%= s.getSid()%></td>
<td><%= s.getName() %></td>
<td><%= s.getAge() %></td>
<td><%= s.getEmail() %></td>
</tr>
</TABLE></center>
<%@ include file="SearchStudentForm.html" %>
</body>
```

What is the difference between Java Persistence API (JPA) and Hibernate ORM Framework ?

JPA and Hibernate Difference

JPA is a specification for accessing, persisting and managing the data between Java objects and the relational database. As the definition says its API, it is only the specification. There is no implementation for the API. JPA specifies the set of rules and guidelines for developing the interfaces that follows standard. **Straight to the point : JPA is just guidelines to implement the Object Relational Mapping (ORM) and there is no underlying code for the implementation**.

Where as, Hibernate is the actual implementation of JPA guidelines. When hibernate implements the JPA specification, this will be certified by the JPA group upon following all the standards mentioned in the specification. For example, JPA guidelines would provide information of mandatory and optional features to be implemented as part of the JPA implementation.

Hibernate is a JPA provider. When there is new changes to the specification, hibernate would release its updated implementation for the

JPA specification. Other popular JPA providers are Eclipse Link (Reference Implementation), OpenJPA, etc

How To Create Session Factory in Hibernate 4 ?

There are many APIs deprecated in the hibernate core framework. One of the frustrating point at this time is, hibernate official documentation is not providing the clear instructions on how to use the new API and it stats that the documentation is in complete. Also each incremental version gets changes on some of the important API. One such thing is the new API for creating the session factory in Hibernate 4.3.0 on wards. In our earlier versions we have created the session factory as below:

SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();

The method buildSessionFactory is deprecated from the hibernate 4 release and it is replaced with the new API. If you are using the hibernate 4.3.0 and above, your code has to be:

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

2 StandardServiceRegistryBuilder builder = new StandardServiceRegistryBuilder().

3 applySettings(configuration.getProperties());

4 SessionFactory factory = configuration.buildSessionFactory(builder.build());

Class ServiceRegistryBuilder is replaced by StandardServiceRegistryBuilder from 4.3.0. It looks like there will be lot of changes in the 5.0 release. Still there is not much clarity on the deprecated

APIs and the suitable alternatives to use. Every incremental release comes up with more deprecated API, they are in way of fine tuning the core framework for the release 5.0.

Hibernate-Common Errors

Problem

org.hibernate.connection.C3P0ConnectionProvider" is missing?

Problem

Configured Hibernate to use "c3p0" connection pool, but hits following warning Look like "org.hibernate.connection.C3P0ConnectionProvider" is missing?

Solution

Since Hibernate v3.3 (if not mistaken), the "C3P0ConnectionProvider" is moved to another jar file "hibernate-c3p0.jar". You need to include it, in order to make Hibernate supports c3p0 connection pool.

You can download the "hibernate-c3p0.jar" from JBoss public repository.

Hibernate – The type AnnotationConfiguration is deprecated

Problem

Working with Hibernate 3.6, noticed the previous "org.hibernate.cfg.AnnotationConfiguration", is marked as "deprecated".

Code snippets …

```
import org.hibernate.cfg.AnnotationConfiguration;

//...

private static SessionFactory buildSessionFactory() {

  try {

  return new AnnotationConfiguration().configure().buildSessionFactory();

    } catch (Throwable ex) {

  System.err.println("Initial SessionFactory creation failed." + ex);

    throw new ExceptionInInitializerError(ex);

         }

}
```

The code is still working, just keep displaying the deprecated warning message, is there any replacement for "AnnotationConfiguration" ?

Solution

In Hibernate 3.6, "org.hibernate.cfg.AnnotationConfiguration" is deprecated, and all its functionality has been moved to "org.hibernate.cfg.Configuration".

So , you can safely replace your "AnnotationConfiguration" with "Configuration" class.

Code snippets …

```java
import org.hibernate.cfg.Configuration;

//...

private static SessionFactory buildSessionFactory() {

 try {

 return new Configuration().configure().buildSessionFactory();

  } catch (Throwable ex) {

 System.err.println("Initial SessionFactory creation failed." + ex);

  throw new ExceptionInInitializerError(ex);

          }

}
```

3) java.lang.ClassNotFoundException : javassist.util.proxy.MethodFilter

Problem

Using Hibernate 3.6.3, but hits this javassist not found error, see below for error stacks :

Solution

javassist.jar is missing, and you can get the latest from JBoss Maven repositorty.

4) org.hibernate.AnnotationException: Unknown Id.generator

Problem

Runing the following Hibernate's annotation sequence generator with PostgreSQL database.

@Id

@Column(name="user_id", nullable=false)

@GeneratedValue(strategy = GenerationType.SEQUENCE ,generator="account_user_id_seq")

private Integer userId;

Hits the following Unknown Id.generator exception.

Caused by: org.hibernate.AnnotationException: Unknown Id.generator: account_user_id_seq

    at org.hibernate.cfg.BinderHelper.makeIdGenerator(BinderHelper.java:413)

    at org.hibernate.cfg.AnnotationBinder.bindId(AnnotationBinder.java:1795)

    at org.hibernate.cfg.AnnotationBinder.processElementAnnotations(AnnotationBinder.java:1229)

    at org.hibernate.cfg.AnnotationBinder.bindClass(AnnotationBinder.java:733)

The sequence "account_user_id_seq" is created in PostgreSQL database, what caused the above exception?

Solution

When declaring the Hibernate's annotation strategy to use "Sequences" as Id generator, try specify the @SequenceGenerator as well, as following

@Id

@Column(name="user_id", nullable=false)

@SequenceGenerator(name="my_seq", sequenceName="account_user_id_seq")

@GeneratedValue(strategy = GenerationType.SEQUENCE ,generator="my_seq")

private Integer userId;

5)Hibernate Error – Initial SessionFactory creation failed.java.lang.NoClassDefFoundError: org/dom4j/DocumentException

A common Hibernate's error, this is caused by the missing dependency library – dom4j.

Solution

You can download the library here – http://sourceforge.net/projects/dom4j/files/dom4j/

6)Hibernate Error – Initial SessionFactory creation
failed.java.lang.NoClassDefFoundError:
org/apache/commons/logging/LogFactory

A common Hibernate's error, this is caused by the missing dependency
library – commons-logging.jar

Solution

You can download the library here – http://commons.apache.org/logging/

7)Hibernate Error – Initial SessionFactory creation
failed.java.lang.NoClassDefFoundError:
org/apache/commons/collections/SequencedHashMap

A common Hibernate's error, this is caused by the missing dependency
library – commons-collections.jar

Solution

You can download the library here –
http://commons.apache.org/collections/

8)Hibernate Error – Initial SessionFactory creation
failed.java.lang.NoClassDefFoundError: net/sf/cglib/proxy/CallbackFilter

A common Hibernate's error, this is caused by the missing dependency
library – cglib.

Solution

You can download the library here – http://cglib.sourceforge.net/

9)Hibernate Error – Initial SessionFactory creation failed.java.lang.NoClassDefFoundError: com/mchange/v2/c3p0/DataSources

Hibernate's "C3P0" connection pool error, this is caused by the missing dependency library – C3P0.

Solution

You can download the library here – http://www.mchange.com/projects/c3p0/

10)Hibernate Error – Initial SessionFactory creation failed.java.lang.NoClassDefFoundError: org/hibernate/annotations/common/reflection/ReflectionManager

This is caused by missing of the Hibernate commons annotations library.

Solution

You can download the library from Hibernate official website

11)Hibernate Error – Exception in thread "main" java.lang.NoClassDefFoundError: antlr/ANTLRException

This is caused by missing of the antlr library. It's usually happened when you did invoke Hibernate's query statement.

Solution

You can download the library from Antlr official website

12)Hibernate Error – java.lang.NoClassDefFoundError: javax/transaction/Synchronization

Problem

This is caused by missing of the "jta.jar", usually happened in Hibernate transaction development.

Solution

You can download "jta.jar" from default Maven central, JBoss or Java.net repositories.

13)java.lang.ClassFormatError : Absent Code attribute in method that is not native or abstract in class file …

Problem

A very strange and rare problem, happened in JPA or Hibernate development.

Solution

This is always caused by the javaee.jar which is located at Java.net. Many developers like to grab the javaee.jarwith the following Maven coordinate :

14)java.lang.NoSuchMethodError: org.objectweb.asm.ClassWriter

Problem

In Hibernate development, it's common to hits the following error message.

Solution

The "Unable to instantiate default tuplizer [org.hibernate.tuple.entity.PojoEntityTuplizer" is a generic error message, it may caused by many reasons. So, you have to look at the last line that caused the error.

Caused by: java.lang.NoSuchMethodError: org.objectweb.asm.ClassWriter

The main cause is the old asm.jar library e.g 'asm-1.5.3.jar', just upgrade the asm library to the most recent version will get rid of the error message. e.g, 'asm-3.1.jar'.

15) java.lang.ClassNotFoundException: javax.persistence.Entity

Problem

In JPA or Hibernate development, it hits the following error message :

Solution

The javax.persistence.Entity is a class inside the J2EE SDK library "javaee.jar", you are missing this jar file in your project classpath.

1. J2EE SDK

You can always get the javaee.jar from http://java.sun.com/javaee/. Download and install the SDK in your computer, the javaee.jar can be found in the "\J2EE_SDK_FOLDER\lib" folder. For example,

C:\Sun\SDK\lib\javaee.jar

Get the javaee.jar file and include it in your project classpath.

16) java.lang.ClassNotFoundException: javax.transaction.TransactionManager

Problem

In JPA or Hibernate development, it hits the following error message :

Solution

The javax.transaction.TransactionManager is a class inside the J2EE SDK library "javaee.jar", you are missing this jar file in your project classpath.

Hibernate Exception – org/dom4j/DocumentException
If you are setting up the first hibernate application, there is chance that you would get the following exception if the required jar is missed. The reason is you have not included the **dom4j-1.6.1.jar** file which is used internally by the hibernate core for parsing the XML documents. The result will be the following exception.

To resolve this exception we can add the the **dom4j-1.6.1.jar file in the classpath.**

Hibernate Exception – javassist.util.proxy?

f you are using Hibernate 4, you would get this exception while setting up the first hibernate application. The missing library is javassist. To resolve this problem set the classpath for the javassist   JAR file .