**HIBERNATE**

What is Hibernate – Hibernate Introduction => In this tutorial am going to explain, why Hibernate came into picture though we have JDBC for connecting to the database, and what is this hibernate frame work first let us see what are the draw backs of JDBC.

## ****Draw Backs of JDBC:****

* In JDBC, if we open a database connection we need to write in try, and if any exceptions occurred catch block will takers about it, and finally used to close the connections.
* here as a programmer we must close the connection, or we may get a chance to get our of connections message…!
* Actually if we didn’t close the connection in the finally block, then jdbc doesn’t responsible to close that connection.
* In JDBC we need to write Sql commands in various places, after the program has created if the table structure is modified then the JDBC program doesn’t work, again we need to modify and compile and re-deploy required, which is tedious.
* JDBC used to generate database related error codes if an exception will occurs, but java programmers are unknown about this error codes right.
* In the Enterprise applications, the data flow with in an application from class to class will be in the form of objects, but while storing data finally in a database using JDBC then that object will be converted into text.  Because JDBC doesn’t transfer objects directly.

In order to overcome above problems,  Hibernate came into picture..!

## 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**.
* Hibernate is a non-invasive framework,  means it wont 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 with out server, i mean it will suitable for all types of java applications (stand alone or desktop or any servlets bla bla.)
* Hibernate is purely for persistence (to store/retrieve data from Database).

## Mapping And Configuration Files In Hibernate

Mapping and Configuration are very familiar keywords we used to here in the hibernate, every hibernate program must need these 2 xml files.

## Mapping:

* Mapping file is the heart of hibernate application.
* Every ORM tool needs this mapping, mapping is the mechanism of placing an object properties into column’s of a table.
* Mapping can be given to an ORM tool either in the form of an XML or in the form of the annotations.
* The mapping file contains mapping from a pojo class name to a table name and pojo class variable names to table column names.
* While writing an hibernate application, we can construct one or more mapping files, mean a hibernate application can contain any number of  mapping files.

generally an object contains 3 properties like

* Identity (Object Name)
* State (Object values)
* Behavior (Object Methods)

But while storing an object into the database, we need to store only the values(State) right ? but how to avoid identity, behavior.. its not possible. In order to inform what value of an object has to be stored in what column of the table, will be taking care by the mapping,  actually mapping can be done using 2 ways,

* XML
* Annotations.

Actually annotations are introduced into java from JDK 1.5.

## ****Syntax Of Mapping xml:****

<hibernate-mapping>

<class name="POJO class name" table="table name in database">

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

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

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

</class>

</hibernate-mapping>

## Configuration:

Configuration is the file loaded into an 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 here, actually in hibernate 1, 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.user">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="dialet">Database dialet class</property>

<property name="hbm2ddl.auto">create/update or what ever</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>

But XML files are always recommended to work.

## Main Advantage And Disadvantages Of Hibernates

## Advantages of hibernates:

* Hibernate supports Inheritance, Associations, Collections
* In hibernate if we save the derived class object,  then its base class object will also be stored into the database, it means hibernate supporting inheritance
* Hibernate supports relationships like One-To-Many,One-To-One, Many-To-Many-to-Many, Many-To-One
* This will also supports collections like List,Set,Map (Only new collections)
* In jdbc all exceptions are checked exceptions, so we must write code in try, catch and throws, but in hibernate we only have Un-checked exceptions, so no need to write try, catch, or no need to write throws.  Actually in hibernate we have the translator which converts checked to Un-checked ;)
* Hibernate has capability to generate primary keys automatically while we are storing the records into database
* Hibernate has its own query language, i.e hibernate query language which is database independent
* So if we change the database, then also our application will works as HQL is database independent
* HQL contains database independent commands
* While we are inserting any record, if we don’t have any particular table in the database, JDBC will rises an error like “View not exist”, and throws exception, but in case of hibernate, if it not found any table in the database this will create the table for us ;)
* Hibernate supports caching mechanism by this, the number of round trips between an application and the database will be reduced, by using this caching technique an application performance will be increased automatically.
* Hibernate supports annotations, apart from XML
* Hibernate provided Dialect classes, so we no need to write sql queries in hibernate, instead we use the methods provided by that API.
* Getting pagination in hibernate is quite simple.

## Disadvantages of hibernates:

* I don’t think there are disadvantages in hibernate
* You know some thing.., Its saying hibernate is little slower than pure JDBC, actually the reason being hibernate used to generate many SQL statements in run time, but i guess this is not the disadvantage :-)
* But there is one major disadvantage, which was boilerplate code issue, actually we need to write same code in several files in the same application, but spring eliminated this.

## Steps To Use Hibernate In Any Java 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…what ever, 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).

## Follow The Steps:

**1.** Import the hibernate API, they are many more, but these 2 are more than enough…

import org.hibernate.\*;  
import org.hibernate.cfg.\*;

**2.** Among Configuration, Mapping xml files, first we need to load configuration xml, because once we load the configuration file, automatically mapping file will be loaded as we registered this mapping xml in the configuration file.

So to load configuration xml, we need to create object of **Configuration** class, which is given in **org.hibernate.cfg.\***;  and we need to call **configure()** method in that class, by passing xml configuration file name as parameter.

Eg:

Configuration **cf** = new Configuration();  
cf.configure(“hibernate.cfg.xml”);

Here our configuration file name is your choice, but by default am have been given hibernate.cfg.xml,  so once this configuration file is loaded in our java app, then we can say that hibernate environment is started in our program.

So once we write the line\_ **cf.configure(“hibernate.cfg.xml”),** configuration object **cf** will reads this xml file **hibernate.cfg.xml,** actually internally cf will uses DOM parsers to read the file.

Finally…

* cf will reads data from hibernate.cfg.xml
* Stores the data in different variables
* And finally all these variables are grouped and create one high level hibernate object we can call as SessionFactory object.
* So Configuration class only can create this SessionFactory object

Like SessionFactory **sf** =  cf.buildSessionFactory();

Actually SessionFactory is an interface not a class, and SessionFactoryImpl is the implimented class for SessionFactory, so we are internally creating object of SessionFactoryImpl class and storing in the interface reference, so this SessionFactory object **sf** contains all the data regarding the configuation file so we can call **sf** as heavy weight object.

3. Creating an object of session,

* Session is an interface and SessionImpl is implemented class, both are given in org.hibernate.\*;
* When ever session is opened then internally a database connection will be opened, in order to get a session or open a session we need to call openSession() method in SessionFactory, it means SessionFactory produces sessions.

Session **session** = **sf**.openSession();

**sf** = SessfionFactory object

4. Create a logical transaction

While working with insert, update, delete, operations from an hibernate application onto the database then hibernate needs a logical Transaction, if we are selecting an object from the database then we do not require any logical transaction in hibernate.  In order to begin a logical transaction in hibernate then we need to call a method beginTransaction() given by Session Interface.

Transaction tx = **sessio**n.beginTransaction();

**session** is the object of Session Interface

5. Use the methods given by Session Interface,  to move the objects from application to database and  from database to application

|  |  |  |
| --- | --- | --- |
| session .save(s) | - | Inserting object ‘**s**‘ into database |
| session.update(s) | - | Updating object ‘**s**‘ in the database |
| session.load(s) | - | Selecting object ‘**s**‘ object |
| session.delete(s) | - | Deleting object ‘**s**‘ from database |

* So finally we need to call **commit()** in Transaction, like **tx.commit()**;
* As i told earlier,  when we open session a connection to the database will be created right, so we must close that connection as session. close().
* And finally close the SessionFactory as **sf.close()**
* That’s it.., we are done.

Final flow will be Configuration 🡪 SessionFactory🡪Session🡪Transaction🡪Close Statements

--------------------------------------------------------------------------------------------------------------------------------------------------

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.setProName("iPhone");

p.setPrice(25000);

Transaction tx = session.beginTransaction();

session.save(p);

System.out.println("Object saved successfully.....!!");

tx.commit(); session.close(); factory.close();

---------------------------------------------------------------------------------------------------------------------------------------------

Object o=session.load(Product.class,new Integer(101));

Product s=(Product)o;

// For loading Transaction scope is not necessary...

System.out.println("Loaded object product name is\_\_\_"+s.getProName());

Object o=session.load(Product.class,new Integer(103));

        Product p=(Product)o;

        Transaction tx = session.beginTransaction();

        session.delete(p);

his is the program to update an object (1 complete row) in the database, which is already persisted in the database, then we have the following two approaches…

## Approach 1 🡪 Load that object from the database, and modify its values, now hibernate automatically modifies the values on to database also, when ever the transaction is committed.

## Approach 2 🡪 If we want to modify object in the database, then create new object with same id and we must call update() given by session interface.

Approach 🡪 1)

Object o=session.load(Product.class,new Integer(105));

Product s=(Product)o;

Transaction tx = session.beginTransaction();

//s.setStno(105);   should not update, because we loaded with that number right..?

s.setPrice(4000);   // implicitly update method will be called..

tx.commit();

 When ever an object is loaded from the database then hibernate stores the loaded object in cache-memory maintained by session-interface

 Once an object is loaded, if we do any modifications on that object by calling its setter methods, then these modification are stored in the object maintained by cache-memory

 if we modify the loaded object for multiple times then also the modifications will be stored in object maintained by the cache-memory only.

 when ever we issue commit() operation then hibernate verify whether any changes are there between the object stored in the cache and object in the database, if changes exists then hibernate automatically updates the database by generating any update operation.

 What am saying is hibernate automatically maintains synchronization between cache-memory object and database table objects (rows)

Approach 🡪 2)

Product p=new Product();

p.setProductId(104);  // 104 must be in the DB

p.setProName("Someting");

Transaction tx = session.beginTransaction();

session.update(p);

tx.commit();

Here we no need to load an object from the database . we will create a new object, and we will assign same id no’s to it and we will call update() explicitly in order to make the changes on the object that is stored in the database

That’s it, actually first approach is  recommended always..

## Hibernate Versioning Example, Hibernate Versioning Of Objects

Once an object is saved in a database, we can modify that object any number of times right, If we want to know how many no of times that an object is modified then we need to apply this versioning concept.  
When ever we use versioning then hibernate inserts version number as **zero**, when ever object is saved for the first time in the database.  Later hibernate increments that version no by one automatically when ever a modification is done on that particular object.  
In order to use this versioning concept, we need the following two changes in our application.

* Add one property of type int in our pojo class as 🡪 private int v ;
* In hibernate mapping file, add an element called version soon after id element. 🡪 <version name="v" column="ver" />
* In this above mapping file, find the <version> element, there i have given column name as **version**, actually you can write any name its not predefined.
* Remember friends, first we must run the logic to **save** the object then hibernate will inset 0 (Zero) by default in the version column of the database, its very important point in the interview point of view also
* First save logic to let the hibernate to insert zero in the version column, then any number of update logic’s (programs) we run, hibernate will increments +1 to the previous value
* But if we run the update logic for the first time, hibernate will not insert zero..! it will try to increment the previous value which is NULL in the database so we will get the exception.
* Guys your know some thing.., actually we can run any logic (Save or Update) :-)for the first time, but make sure the versioning column is a number (>=0), but save logic has ability to insert zero by default if there is no value, and update logic will directly tries to increments already existing value by 1, it wont insert any value by default if its null.

# Importance Of Wrapper And Primitive Types In Hibernate

# Primitive type insert zero in database whereas wrapper type insert null blank value in db .

## Hibernate Lifecycle Of pojo Class Objects

Actually our POJO class object having 3 states like…

* Transient state
* Persistent state
* Detached state

## Transient & Persistent states:

* When ever an object of a pojo class is created then it will be in the Transient state
* When the object is in a Transient state it doesn’t represent any row of the database, i mean not associated with any Session object, if we speak more we can say no relation with the database its just an normal object
* If we modify the data of a pojo class object, when it is in transient state then it doesn’t effect on the database table
* When the object is in persistent state, then it represent one row of the database, if the object is in persistent state then it is associated with the unique Session
* if we want to move an object from persistent to detached state, we need to do either closing that session or need to clear the cache of the session
* if we want to move an object from persistent state into transient state then we need to delete that object permanently from the database

Configuration cfg = new Configuration();

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

SessionFactory factory = cfg.buildSessionFactory();

Session session = factory.openSession();

         // Transient state\_\_\_\_\_start

Product p=new Product();

p.setProductId(101);

p.setProName("iPhone");

p.setPrice(25000);

         // Transient state\_\_\_\_\_end

         // Persistent state\_\_\_\_\_start

Transaction tx = session.beginTransaction();

session.save(p);

System.out.println("Object saved successfully.....!!");

tx.commit();

         // Persistent state\_\_\_\_\_end

session.close();

factory.close();

if we want to convert the object from Transient state to Persistentstate we can do in 2 ways

* By saving that object like above
* By loading object from database

If we do any modifications all the changes will first applied to the object in session cache only (Let\_\_ we do the modifications 5 times, then 5 times we need to save the changes into the database right, which means number of round trips from our application to database will be increased, Actually if we load an object from the database, first it will saves in the cache-memory so if we do any number of changes all will be effected at cache level only and finally we can call save or update method so with the single call of save or update method the data will be saved into the database.

If we want to save an object into database then we need to call any one of the following 3 methods

* save()
* persist()
* saveOrUpdate()

i will explain about persist, saveOrUpdate methods later….

If we want to load an object from database, then we need to call either load() or get() methods

## Transient:

One newly created object,with out having any relation with the database, means never persistent, not associated with any Session object

## Persistent:

Having the relation with the database, associated with a unique Session object

## Detached:

previously having relation with the database [persistent ], now not associated with any Session

## Hibernate Converting Object From Detached to Persistent state

 Configuration cfg = new Configuration();

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

        SessionFactory factory = cfg.buildSessionFactory();

        Session session1 = factory.openSession();

         Product p=null;          //Transient state..

         Object o=session1.get(Product.class, new Integer(1001));

         p=(Product)o;           //now p is in Persistent state..

        session1.close();

        p.setPrice(36000);            // p is in Detached state

        Session session2=factory.openSession();

         Transaction tx=session2.beginTransaction();

            session2.update(p);      // now p reached to Persistent state

         tx.commit();

        session2.close();

        factory.close();

* We have opened the session1 at line number **14** and closed at line number **20**, see i have been loaded the Product class object by using get(-,-) method
* **Remember** get() method always returns the super class object (Object)
* so i typecast into my pojo class object type, so now we can use print any value from this object so its in the Persistent state
* see line number **22**,  am trying to change the Price,  but it wont effect the database because its not in the session cache so i need to take one more session to update this value in the database, so for that reason i took one more session from line numbers 24 – 30
* Gun short point is, things related to database must go with in the session only that’s it

## Hibernate Inheritance: Table Per Class Hierarchy

Here is the explanation and one example on hibernate table per class hierarchy, consider we have base class named Payment and 2 derived classes like CreditCard, Cheque.

If we save the derived class object like CreditCard or Cheque then automatically Payment class object will also be saved into the database, and in the database all the data will be stored into a **single table** only, which is base class table for sure. But here we must use one extra discriminator column in the database,  just to identify which derived class object we have been saved in the table along with the base class object,  if we are not using this column hibernate will throws the exception, see this example so that you will get one idea on this concept.

Payment 🡪 int paymentId , double amount.

creaditCard 🡪string creaditcardTYpe

cheque 🡪 string chequetype

Configuration cfg = new Configuration();

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

        SessionFactory factory = cfg.buildSessionFactory();

        Session session = factory.openSession();

        CreditCard c=new CreditCard();

        c.setPaymentId(10);

        c.setAmount(2500);

        c.setCreditCardType("Visa");

        Cheque c1=new Cheque();

        c1.setPaymentId(11);

        c1.setAmount(2600);

        c1.setChequeType("ICICI");

        Transaction tx = session.beginTransaction();

        session.save(c);

        session.save(c1);

        System.out.println("Object saved successfully.....!!");

        tx.commit();

        session.close();

        factory.close();

## Payment.hbm.xml:

<hibernate-mapping>

<class name="str.Payment" table="PAYMENT">

<id name="paymentId" column="pid" />

<discriminator column="dcolumn" type="string" length="5"/>

<property name="amount" column="amt" />

<subclass name="str.CreditCard" discriminator-value="CC">

<property name="CreditCardType" column="cctype" length="10" />

</subclass>

<subclass name="str.Cheque" discriminator-value="cq">

<property name="ChequeType" column="cqtype" length="10" />

</subclass>

</class>

</hibernate-mapping>

* Payment.java, CreditCard.java, Cheque.java are just pojo classes nothing to explain, but see in CreditCard.java, Cheque.java i have inherited the Payment.java.
* In this inheritance concept, mapping file is the central part, see in line number **10**,  we added one new line  discriminator, after the id element just to identify which derived class object we have been saved in the table (see the oracle console once)
* every thing has been saved in a single table

## Hibernate Inheritance: Table Per subClass Hierarchy

This is also just like previous example, but some changes are there, in table per class hierarchy all the data was saved in a single table but here,

🡪 x number of classes = x number of tables in the database.

If we save the CreditCard class object, then first hibernate will saves the data related to super class object into the super class related table in the database and then CreditCard object data in CreditCard related table in the database, so first base class data will be saved

<hibernate-mapping>

<class name="str.Payment" table="PAYMENT">

<id name="paymentId" column="pid" />

<property name="amount" column="amt" />

<joined-subclass name="str.CreditCard" table="CreditCard">

<key column="dummy1" />

<property name="CreditCardType" column="cctype" length="10" />

</joined-subclass>

<joined-subclass name="str.Cheque" table="Cheque">

<key column="dummy2" />

<property name="ChequeType" column="cqtype" length="10" />

</joined-subclass>

</class>

</hibernate-mapping>

In the mapping file,  <key –> element is because,  once we save the derived class object, then hibernate will first save the baseclass object then derived class object right ..!, so at the time of saving the derived class object hibernate will copy the primary key value of the base class into the corresponding derived class, see in the above output 10 copied into dummy1 column of CreditCard table and 11 copied into Dummy2 column of the cheque table.

## Hibernate Inheritance: Table Per Concrete Class Hierarchy

Something like previous example but the changes are at mapping file only, and one more thing is..

🡪 x number of derived classes = x number of tables in the database

* Once we save the derived class object, then derived class data and base class data will be saved in the derived class related table in the database
* for this type we need the tables for derived classes, but not for the base class
* in the mapping file we need to use one new element <union-subclass — >under <class —>

<hibernate-mapping>

<class name="str.Payment" table="PAYMENT">

<id name="paymentId" column="pid" />

<property name="amount" column="amt" />

<union-subclass name="str.CreditCard">

<property name="CreditCardType" column="cctype" length="10" />

</union-subclass>

<union-subclass name="str.Cheque">

<property name="ChequeType" column="cqtype" length="10" />

</union-subclass>

</class>

</hibernate-mapping>

## Example On Composite Primary Keys In Hibernate

Composite primary keys means having more than one primary key, let us see few points on this concept

* If the table has a primary key then in the hibernate mapping file we need to configure that column by using <id /> element right..!
* Even though the database table doesn’t have any primary key, we must configure one column as id (one primary key is must)
* If the database table has more than one column as primary key then we call it as composite primary key, so if the table has multiple primary key columns , in order to configure these primary key columns in the hibernate mapping file we need to use one new element called **<composite-id** …..> **</composite-id>**

public class Product implements java.io.Serializable{

private int productId;

private String proName;

private double price;

------------------------------------------------------------------------------------------

|  |
| --- |
| <hibernate-mapping>  <class name="str.Product" table="products">  <composite-id>  <key-property name="productId" column="pid"  />  <key-property name="proName" column="pname" length="10" />  </composite-id>  <property name="price"/>  </class>  </hibernate-mapping> |

-----------------------------------------------------------------------

SessionFactory factory = cfg.buildSessionFactory();

Session session = factory.openSession();

Product p=new Product();

p.setProductId(101);

p.setProName("iPhone");

p.setPrice(25000);

Transaction tx=session.beginTransaction();

session.save(p);

System.out.println("Object Loaded successfully.....!!");

tx.commit();

* see Product.java pojo class, in line number 3 i have implemented the **java.io.Serializable** interface,  this is the first time am writing this implementation for the pojo class right…!  we will see the reason why we use this serializable interface later.
* But remember, if we want to use the composite primary keys we must implement our pojo class with **Serializable** interface
* **hibernate.cfg.xml** is normal as previous programs, something like hello world 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>
* Actually we will see the exact concept of this composite primary keys in the next example (loading an object with composite key)

## Generators <generator> In Hibernate

<generator /> is one of main element we are using in the hibernate framework [in the mapping file],  let us see the concept behind this generators.

* Up to now in our hibernate mapping file, we used to write <generator /> in the id element scope, actually this is default like whether you write this assigned generator or not hibernate will takes automatically
* In fact this assigned means hibernate will understand that, while saving any object hibernate is not responsible to create any primary key value for the current inserting object, user has to take the response
* The thing is, 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 class is created by hibernate for its implementation
* hibernate provided different primary key generator classes and all these classes are implemented from

org.hibernate.id.IdentifierGeneratar Interface

while configuring <generator /> element in mapping file, we need to pass parameters if that generator class need any parameters, actually one sub element of <generator /> element is <param />, will talk more about this

<generator class="">

<param name=""> value </param>

</generator>

## List of generators

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

* assigned
* increment
* sequence
* identity
* hilo
* native
* foregin
* uuid.hex
* uuid.string

In the above generators list, the first **7** are used for int,long,short types of primary keys, and last **2** are used when the primary key column type is String type (varchar2)

## 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”
* If generator class is assigned, then the programmer is responsible for assigning the primary key value to object which is going to save into the database

<id name="prodId" column="pid">

<generator />

</id>

## Increment

* This generator supports in all the databases, database independent
* 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 we manually assigned the value for primary key for an object, then hibernate doesn’t considers that value and uses **max value of id in database + 1** concept only :-)
* If there is no record initially in the database, then for the first time this will saves primary key value as 1, as…

max of id value in database + 1  
0 + 1  
result -> 1

## sequence

* Not has the support with MySql
* This generator class is database dependent it means, we cannot use this generator class for all the database, we should know whether the database supports sequence or not before we are working with it
* while inserting a new record in a database, hibernate gets next value from the sequence under assigns that value for the new record
* If programmer has created a sequence in the database then that sequence name should be passed as the generator

<id name="productId" column="pid">

<generator>

<param name="sequence">MySequence</param>

</genetator>

</id>

* If the programmer has not passed any sequence name, then hibernate creates its own sequence with name “**Hibernate-Sequence**” and gets next value from that sequence, and than assigns that id value for new record
* But remember, if hibernate want’s to create its own sequence, in hibernate configuration file, **hbm2ddl.auto** property must be set enabled

sql> create sequence MySequence incremented by 5;

* first it will starts with 1 by default
* though you send the primary key value., hibernate uses this sequence concept only
* But if we not create any sequence, then first 1 and increments by 1..bla bla. in this case hibernate creating right..? so ensure we have hbm2ddl.auto enabled in the configuration file

## identity

* This is database dependent, actually its not working in oracle
* In this case (identity generator) the id value is generated by the database, but not by the hibernate, but in case of increment hibernate will take over this
* this identity generator doesn’t needs any parameters to pass
* this identity generator is similar to increment generator, but the difference was increment generator is database independent and hibernate uses a select operation for selecting max of id before inserting new record
* But in case of identity, no select operation will be generated in order to insert an id value for new record by the hibernate

<id name="productid" column="pid">

<generator class="......."/>

</id>

## hilo

* This generator is database independent
* 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”**
* if we want to modify the table and column names theen wee need to pass 2 parameter’s for the hilo generators

<id name="productId" column="pid">

<generator>

<param name="table">your table name</param>

<param name="column">your column name </param>

</generator>

</id>

## native

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

## foreign

we will see about this generator in one-to-one relationship, else you may not understand.

## Criteria Query

Unlike HQL, Criteria is only for selecting the data from the database, that to we can select complete objects only not partial objects, in fact by combining criteria and projections concept we can select partial objects too we will see this angle later,  ;)but for now see how we are using criteria for selecting complete objects form the database. We cant perform non-select operations using this criteria.  Criteria is suitable for executing dynamic queries too, let us see how to use this criteria queries in the hibernate.

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

// let Product is our pojo class

List l = crit.list()

// need to call list() to execute criteria

Iterator it = l.iterator();

while(it.hasNext())

{

Object o = it.next();

Product p = (Product)o;

------ ----- -----

}

## Adding Conditions To Criteria

* If we want to put conditions to load data from database, using criteria then we need to create one Criterion Interface object and we need to add this object to Criteria Class object
* Criterion is an interface given in “org.hibernate.criterion” package
* In order to get Criterion object, we need to use Restrictions class
* Restrictions is the factory for producing Criterion objects, but friends there is no explicit relation between Criterion interface and Restrictions class, it means Restrictions class is not implemented from Criterion Interface
* In Restrictions class, we have all static methods and each method of this class returns Criterion object
* Restrictions class is also given in “org.hibernate.criterion” package

Criteria crit = session.createCriteria(Products.class);

Criterion c1=Restrictions.gt("price", new Integer(12000));

//price is our pojo class variable

crit.add(c1); // adding criterion object to criteria class object

List l = crit.list(); // executing criteria query

Note: See line number 2, am calling gt(-,-) method of Restrictions class, (means greater than), in our above example am fetching the data by comparing price greater than (>) 12000

If we want to put more conditions on the data (multiple conditions) then we can use **and** method , **or** method give by the Restrictions class

crit.add(Restrictions.and(Restrictions.like("proName","%R%"),

Restrictions.eq("price",new Integer(12000))));

List l=crit.list();

Iterator it = l.iterator();

Configuration cfg = new Configuration();

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

SessionFactory factory = cfg.buildSessionFactory();

Session session = factory.openSession();

Criteria crit = session.createCriteria(Product.class);

Criterion cn = Restrictions.gt("price",new Double(17000));

crit.add(cn);

List l=crit.list();

System.out.println("List total size...\_"+l.size());

Iterator it=l.iterator();

while(it.hasNext())

{

Product p=(Product)it.next();

System.out.println(p.getProductId());

System.out.println(p.getProName());

System.out.println(p.getPrice());

System.out.println("-----------------");

}

session.close();

factory.close();

* In line number 25, created object of Criteria
* In line number 26,  created Criterion interface object by using Restrictions class
* In line number 27, added criterion interface object to criteria object
* In line number 28,  executed criteria query by calling list() method in criteria
* See line number 34, in this case we must typecast into our POJO class type only

Actually the internal concept is,  once we called the list() method in criteria (line number 28) all objects (records) will come and stores in list object, from there we used to take iterate then typecast into our POJO class type bla bla.

## Hibernate Projections Introduction

S o far in criteria, we are able to load complete object right….! let us see how to load the partial objects while working with criteria.  The projections concept is introduced in hibernate 3.0 and mainly we can do the following 2 operations using the projection

* We can load partial object from the database
* We can find the Result of Aggregate functions

*Projection* is an Interface given in “org.hibernate.criterion” package, *Projections* is an class given in same package,  actually Projection is an interface, and Projections is an class and is a factory for producing projection objects.

In Projections class, we have all static methods and each method of this class returns Projection interface object.

If we want to add a Projection object to Criteria then we need to call a method **setProjection()**

**Remember**, while adding projection object to criteria, it is possible to add one object at a time.  It means if we add 2nd projection object then this 2nd one will overrides the first one (first one wont be work), so at a time we can only one projection object to criteria object.

Using criteria, if we want to load partial object from the database, then we need to create a projection object for property that is to be loaded from the database

**Example:**

Criteria crit = session.createCriteria(Products.class);

crit.setProjection(Projections.proparty("proName"));

List l=crit.list();

Iterator it=l.iterator();

while(it.hasNext())

{

String s = (String)it.next();

// ---- print -----

}

If we add multiple projections to criteria then the last projection added will be considered to execute see…

Criteria crit = session.createCriteria(Products.class);

Projection p1 = Projection.property("proName");

Projection p2 = Projection.property("price");

crit.setProjection(p1):

crit.setProjection(p2):

List l=crit.list();

-------------- ----

------- - ---  ----

Here collections list l, is going to contain the price in the form of Double objects, but product names are over ridden,  second projection over rides the first one, i mean p2 only will works p1 will not works, actually there is a way to add multiple projections to criteria to select more than one column from the database i will tell you in the next example :-)

## Difference between HQL and Criteria Query in Hibernate

* HQL is to perform both select and non-select operations on the data,  but Criteria is only for selecting the data, we cannot perform non-select operations using criteria
* HQL is suitable for executing Static Queries, where as Criteria is suitable for executing Dynamic Queries
* HQL doesn’t support pagination concept, but we can achieve pagination with Criteria
* Criteria used to take more time to execute then HQL
* With Criteria we are safe with SQL Injection because of its dynamic query generation but in HQL as your queries are either fixed or parametrized, there is no safe from SQL Injection.

## Hibernate Native SQL Query Example

Native SQL is another technique of performing bulk operations on the data using hibernate

* By using Native SQL, we can perform both select, non-select operations on the data
* In face Native SQL means using the direct SQL command specific to the particular (current using) database and executing it with using hibernate

## 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, some times it makes the hibernate application as database dependent one

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 use a method createSQLQuery() given by session interface.

While executing native sql queries on the database, we use directly tables, column names directly in our command.

**Remember**, while executing Native SQL Queries, even though we are selecting complete objects from teh database we need to type cast into object array only, not into our pojo class type, because we are giving direct table, column names in the Native SQL Querie so it does’nt know our class name

If we execute the command, always first it will put’s data in **ResultSet** and from there List

SQLQuery qry = session.createSQLQuery("select \* from PRODUCTS");

// Here PRODUCTS is the table in the database...

List l = qry.list();

Iterator it = l.iterator();

while(it.hasNext())

{

Object row[] = (Object[])it.next(); }

* while selecting data from the table, even though you are selecting the complete object from the table, in while loop still we type cast into object array only right
* See the above code, we typecast into the object[] arrays right..,  in case if we want to type cast into our POJO class (i mean to get POJO class obj), then we need to go with entityQuery concept
* In order to inform the hibernate that convert each row of ResultSet into an object of the POJO class back, we need to make the query as an entityQuery
* to make the query as an entityQuery, we need to call addEntity() method.

//We are letting hibernate to know our pojo class too

SQLQuery qry = session.createSQLQuery("select \* from PRODUCTs").addEntity(Product.class);

List l = qry.list();

Iterator it = l.iterator();

while(it.hasNext())

{

Product p = (Product)it.next();

--- -------

}

**Notes:**

* See line number 2, i have been added addEntity(Product.class) at the end, which will let the hibernate to know about our POJO class, so now we can typecast into our POJO class type like what i have done at  line number 7
* And that’s it, this is the total concept on this Native SQL, am not going to give any example on this separately hope you understood the concept

## Hibernate Relationships

U sing hibernate, if we want to put relationship between two entities [ objects of two pojo classes ], then in the database tables, there must exist foreign key relationship, we call it as Referential integrity.

The main advantage of putting relation ship between objects is, we can do operation on one object, and the same operation can transfer onto the other object in the database [ remember, object means one row in hibernate terminology ]

While selecting, it is possible to get data from multiple tables at a time if there exits relationship between the tables, nothing but in hibernate relationships between the objects

Using hibernate we can put the following 4 types of relationships

* One-To-One
* Many-To-One
* Many-To-Many
* One-To-Many

## Hibernate Cascade Options – Cascade Attribute In Hibernate

M ain concept of hibernate relations is to getting the relation between parent and child class objects

Cascade attribute is mandatory, when ever we apply relationship between objects, cascade attribute transfers operations done on one object onto its related child objects

If we write cascade = “all” then changes at parent class object will be effected to child class object too,  if we write cascade = “all” then all operations like insert, delete, update at parent object will be effected to child object also

Example: if we apply insert(or update or delete) operation on parent class object, then child class objects will also be stored into the database.

default value of cascade =”none” means no operations will be transfers to the child class

Example: if we apply insert(or update or delete) operation on parent class object, then child class objects will not be effected, if cascade = “none”

Cascade having the values…….

* none (default)
* save
* update
* save-update
* delete
* all
* all-delete-orphan

In hibernate relations, if we load one parent object from the database then child objects related to that parent object will be loaded into one collection right (see one-to-many insert example).

Now if we delete one child object from that collection, then the relationship between the parent object and that child object will be removed, but the record (object) in the database will remains at it is, so if we load the same parent object again then this deleted child will not be loaded [ but it will be available on the database ]

so finally what am saying is all-delete-orphan means, breaking relation between objects not deleting the objects from the database, hope you got what am saying ;)

**Note**:  
what is orphan record ..?  
an orphan record means it is a record in child table but it doesn’t have association with its parent in the application.

[ **And** ]

In an application, if a child record is removed from the collection and if we want to remove that child record immediately from the database, then we need to set the cascade =”all-delete-orphan”

And that’s it about this cascade attribute in hibernate, hope i explained all the values..!!

## 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

## Hibernate Annotations Introduction Gun short point :-  Annotations are replacement for XML

Let us see few points regarding annotations in hibernate

* Annotations are introduced in java along with JDK 1.5, annotations are used to provide META data to the classes, variables, methods of java
* Annotations are given by SUN as replacement to the use of xml files in java
* In hibernate annotations are given to replace hibernate mapping [ xml ] files
* While working with annotations in hibernate, we do not require any mapping files, but hibernate xml configuration file is must
* hibernate borrowed annotations from java persistence API but hibernate it self doesn’t contain its own annotations
* Every annotations is internally an Interface, but the key words starts with @ symbol
* For working with annotations, our java version must be 1.5 or higher and hibernate version must be hibernate 3.3 or higher, actually up to now we have been used hibernate 2.2 in our previous tutorials, get ready to download the higher version :-)
* At j2se level, sun has provided very limited set of annotations like @Override and @Deprecated …etc…
* Sun has provided the annotations related to j2se level under java.lang.annotations.\* package
* Most of the annotations related to j2ee level are given by sun and their implementations are given by the vendors
* In hibernate, the annotations supported are given by sun under javax.persistence package and hibernate has provided implementations for annotations given in that package

The basic annotations we are using while creating hibernate pojo classes are…

* @Entity
* @Table
* @Id
* @Column

Actually @Entity, @Table are class level annotations, and @Id, @Column are the field level annotations, no worries you will be able to understand while seeing the first example :-)

* in hibernate, if we are annotations in the pojo class then hibernate mapping file is not required, it means annotations are reducing the use of xml files in the hibernate

Though we are using annotations in our pojo class with mapping xml also, then hibernate will give first preference to xml only not for annotations,  actually this concept is same in struts, hibernate, spring too

## Difference Between Merge And Update Methods In Hibernate

B oth update() and merge() methods in hibernate are used to convert the object which is in detached state into persistence state.  But there is little difference.  Let us see which method will be used in what situation.

SessionFactory factory = cfg.buildSessionFactory();

Session session1 = factory.openSession();

Student s1 = null;

Object o = session1.get(Student.class, new Integer(101));

s1 = (Student)o;

session1.close();

s1.setMarks(97);

Session session2 = factory.openSession();

Student s2 = null;

Object o1 = session2.get(Student.class, new Integer(101));

s2 = (Student)o1;

Transaction tx=session2.beginTransaction();

session2.merge(s1);

* See from line numbers 6 – 9, we just loaded one object s1 into session1 cache and closed session1 at line number 9, so now object s1 in the session1 cache will be destroyed as session1 cache will expires when ever we say session1.close()
* Now s1 object will be in some RAM location, not in the session1 cache
* here s1 is in detached state, and at line number 11 we modified that detached object s1, now if we call update() method then hibernate will throws an error, because we can update the object in the session only
* So we opened another session [session2] at line number 13,  and again loaded the same student object from the database, but with name s2
* so in this session2, we called **session2.merge(s1)**; now into s2 object s1 changes will be merged and saved into the database

Hope you are clear…, actually update and merge methods will come into picture when ever we loaded the same object again and again into the database, like above.

## Difference Between Hibernate Save And Persist Methods

A ctually the difference between hibernate save() and persist() methods is depends on generator class we are using.

* If our generator class is assigned, then there is no difference between save() and persist() methods. Because generator ‘assigned’ means, as  a programmer we need to give the primary key value to save in the database right [ Hope you know this generators concept ]
* In case of other than assigned generator class, suppose if our generator class name is Increment means hibernate it self will assign the primary key id value into the database right [ other than assigned generator, hibernate only used to take care the primary key id value remember :-)], so in this case if we call save() or persist() method then it will insert the record into the database normally

But here thing is,  save() method can return that primary key id value which is generated by hibernate and we can see it by

long s = session.save(k);

In this same case, persist() will never give any value back to the client, hope you are clear.

## Difference Between Hibernate get() and load() Methods ?

W hat is the difference between hibernate get() and load() methods ? this is one of the famous hibernate interview questions,  most of us will use hibernate get and load methods irrespective of knowing their exact behavior :-)will you

accept that ? of course even myself too in the initial stages :-)Today i will try to clear that confusion.

## Few Points About Hibernate get() & load()

* Both are from Session interface, and we will call them as session.get() & session.load()
* Both will be use for retrieving the object (a row) from the database

Then what’s the difference them ? lets start with load() and then get() method.

Consider a Student class having 3 properties stdId, stdName, stdCountry

## 1. session.load()

* When you call session.load() method, it will always return a “proxy” object,  whats the meaning of proxy object ?
* Proxy means, hibernate will prepare some fake object with given identifier value in the memory without hitting the database, for example if we call session.load(Student.class,new Integer(107));  > hibernate will create one fake Student object [row] in the memory with id 107, but remaining properties of Student class will not even be initialized, observe this graphical representation…
* It will hit the database only when we try to retrieve the other properties of Student object i mean stdName, stdCountry.  If we call s2.getStdName() then hibernate will hit the database and search the row with student id 107 and retrieve the values, if object [row] not found in the database it will throws ObjectNotFoundException.,

Let me explain each point by taking an example

System.out.println("Student is calling with load()");

s2 =(Student) session.load(Student.class,new Integer(107));

System.out.println("Student called with load()");

System.out.println("Printing Student Name\_\_\_"+s2.getStdtId());

System.out.println("Printing Student Name\_\_\_"+s2.getStdName());

In index.jsp > line number 4,  i have called **s2.getStdtId()** and hibernate simply printed 107 [at Output > line number 3] without creating any database query why ? because as i have explained hibernate will prepare some fake object with given identifier value in the memory without hitting the database.  So when we call load() method (at index.jsp > line number 2 ) with 107 value, hibernate will create a fake object with 107 as identifier value, i mean temporarily it will consider the student id as 107.  If you observe the output, hibernate was generated the database query (at Output > line number 4 )  when we called **s2.getStdName()**; ( index.jsp > line number 5 ).

That’s it, so finally we came to know that session.load() will hit the database only when we start retrieving the object (row) values.

## 2. session.get()

* When you call session.get() method, it will hit the database immediately and returns the original object

If the row is not available in the database, it returns null

System.out.println("Student is calling with get()");

s1 = (Student) session.get(Student.class,new Integer(107));

System.out.println("Student called with get()");

System.out.println("Printing Student Name\_\_\_"+s1.getStdtId());

System.out.println("Printing Student Name\_\_\_"+s1.getStdName());

Nothing to explain here :-)as i told, when we call session.get() method hibernate will hit the database and returns the original object [ row ], that’s the reason it was generated a query [ check line number 2 in the output screen]

Hmm…, so which is the best method to use, hibernate load() or get() ? its completely your choice ;)