

# JavaEra.com

A Perfect Place for All **Java** Resources

Core Java | Servlet | JSP | JDBC | Struts | Hibernate | Spring

Java Projects | FAQ's | Interview Questions | Sample Programs

Certification Stuff | eBooks | Interview Tips | Forums | Java Discussions

For More Java Stuff Visit

# www.JavaEra.com

A Perfect Place for All **Java** Resources



Hi Friends!

Welcome to JavaEra.com.

As an **admin**, let me introduce myself to all of you.

I am **Anil Reddy** from Hyderabad. I'm a normal guy who lives life to the fullest, **loves programming** and lives a **happy-go-lucky** kind of life.

### **Why I started JavaEra.com?**

I was a job seeker like you all but I was not sure about how to attempt an interview or various companies selection process. I had searched many Portals on the web & joined many job groups. From all those sources I was getting the information regarding latest job openings only but not any resources to chase job selection process by improving my skills.

**Why we need those skills?** Just go through this small story...

Once upon a time a very strong woodcutter asks for a job in a timber merchant, and he got it. The paid was really good and so were the work conditions. For that reason, the woodcutter was determined to do his best. His boss gave him an axe and showed him the area where he was supposed to work. The first day, the woodcutter brought 18 trees "Congratulations," the boss said. "Go on that way!" Very motivated for the boss' words, the woodcutter try harder the next day, but he only could bring 15 trees. The third day he try even harder, but he only could bring 10 trees. Day after day he was bringing less and less trees. "I must be losing my strength", the woodcutter thought. He went to the boss and apologized, saying that he could not understand what was going on.

"When was the last time you sharpened your axe?" the boss asked. "Sharpen? I had no time to sharpen my axe. I have been very busy trying to cut trees.

*If we are just busy in applying for jobs, when we will sharpen our skills to chase the job selection process?*

I don't have many friends who can suggest me or who can share some tips. I got irritated like anything. I thought that like me, there may be millions of my brothers and sisters across the nation facing same kind of problem.

So I strongly committed to start a special zone for all job seekers & IT professionals where everyone can sharpen their skills to find their dream job & in their dream company.

As a result in 10 months nearly 15000+ job seekers joined our family. Thousands of people got benefited by utilizing our resources.

This is just a start, what we have achieved till now is just 0.1% in our vision which is "*Creating a perfect place to share Java knowledge*".

Now I am not alone in chasing this challenge. We will work together by sharing knowledge & giving a ray of hope to millions of our brothers and sisters across the nation.

#### **My Technical Skills:**

Google SEO (Search Engine Optimization) Certified.

JSE (Certified), Servlets, JSP, Struts, Hibernate, Spring, PHP, Zend Framework.

HTML, JavaScript, Jquery, AJAX, CSS, Adobe Photoshop CS6, Adobe Illustrator CS5, Adobe Flash.

*My aim is to provide good and quality articles and content to readers and visitors to understand easily. All articles are written and practically tested by me before publishing online. If you have any query or questions regarding any article feel free to leave a comment or You can get in touch with me On [anilreddy@JavaEra.com](mailto:anilreddy@JavaEra.com), [www.facebook.com/JavaAnil](http://www.facebook.com/JavaAnil)*

**Regards**

Anil Reddy

Founder & Administrator

JavaEra.com

**Email :**

[anilreddy77466@gmail.com](mailto:anilreddy77466@gmail.com),

[anilreddy@JavaEra.com](mailto:anilreddy@JavaEra.com).

[www.facebook.com/JavaAnil](http://www.facebook.com/JavaAnil)

*Anil  
giriday*

## Persistence :-

- persistence is a term which denotes permanent storage (or) permanent area.
- An application data stored permanently is called persistent data.
- The logic used to make some data as persistent is called as "persistence logic".
- As a Java programmer, we have the following options to make some data as persistent

→ ~~using Flat file, XML file, Database, etc.~~

→ ~~using Flat file, XML file, Database, etc.~~

① A Flat file

② An XML file

③ A Database

- A Flat file is nothing but a file which is not related with any technology (or) any framework.

→ In Java, persistent data means, the data which has more lifetime than the execution of a Java application. It means even though the Java appn execution is terminated but still the data produced (or) generated by the appn is still alive.

→ As a Java programmer, if we ~~use~~ a Backend as a flat file then we have I/O streams for reading (or) writing the data in a file (or) using a file.

→ A Java program use InputStream for read the data from a file, OutputStream for writing the data into a file.

→ with files, we got the following drawbacks.

- ① files are completed complexed work, Bcz we need to write some programming code
- ② for files, we donot have any common query language like Databases.
- ③ files can't be used for storing large amounts of data.
- ④ files are very less secured
- ⑤ we can't maintain integrity b/w the data

## ⑥ XML as a backend :-

→ Using XML as a backend is almost equal to a flat file as a backend. But the difference is a flat file can just store the data, an XML file can store the data and also can describe the data.

Eg:- Flat file

txt		
100	ABC	500
.		
.		

→ It is not descriptive file, just store the data only.

XML

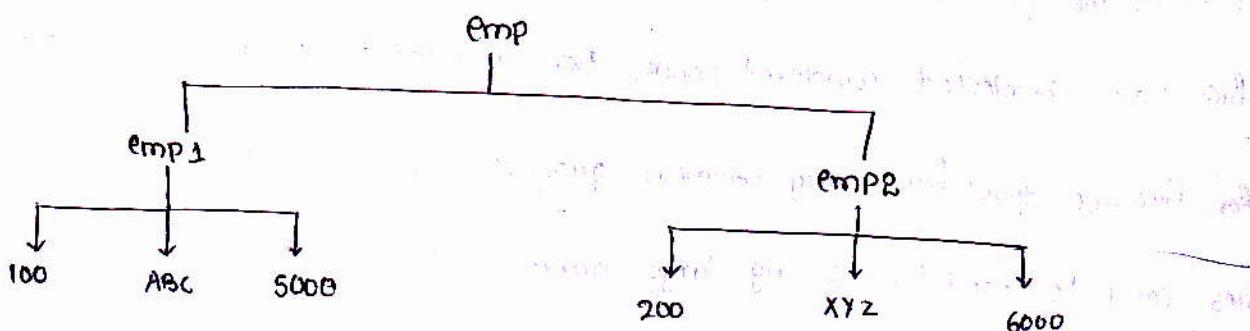
```

<emp>
  <empid> 100 </empid>
  <ename> ABC </ename>
  <sal> 500 </sal>
</emp>
  
```

→ This file store the data and also clearly describe the data.

→ with XML as a backend, we have the following problems

- ① for constructing an XML file we must follow some rules.
- ② for searching for some data, always an XML must be loaded and parsed & then a tree structure is created and then search begins. It means XML is slower than databases.



- ③ XML files are less secured.
- ④ we can't manage integrity b/w the data (or) referential integrity b/w the data

NOTE:- The benefit of XML is, we can transfer the data across Network in Language Independent, platform Independent and Browser Independent.

## \* DataBase as a Backend :-

→ with File Management System we got some drawbacks, so we entered into DataBase management System.

→ with Databases we have the following advantages

① Databases are Simple to work with it.

② we have a common query Language (SQL).

③ The data stored in a DataBase is secured

④ A DataBase can store large amount of data

⑤ we can manage the Integrity b/w the data in a database

etc....

05/02/2013

Q: Why we need to go for Object Relational Mapping (ORM)?

→ Sun micro Systems given JDBC technology for java applications to talk with databases.

→ Even though JDBC is a good technology for connecting with databases. But in a

realtime environment, the technology has produced some problems (or) drawbacks.

① while creating JDBC persistence logic, we keep the DataBase structure in mind and we write the JDBC application code. After the application is completed, if the database structure is modified then we need to make the changes in the entire application code, whenever the JDBC code is added. It makes burden on a programmer. ( $db \rightarrow code$ )

② In applications of realtime, the data will be in Objects & the data will be transferred with in the application, also in the form of objects but when storing it in database

the JDBC technology cannot transfer objects b/w application & a database. So an object is converted to a text & then the text will be transferred to the database

Here the application code is increased for converting ~~of~~ object into a text or text into an object & also the burden on programmer will be increased.

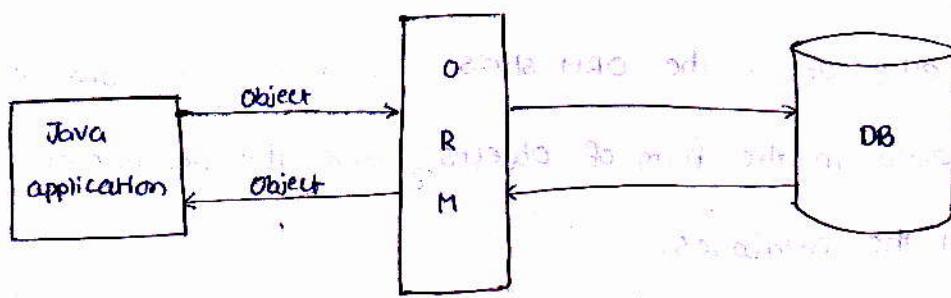
(Objects  $\rightarrow$  Text)

- ③ JDBC technology uses SQL operations & the SQL based persistence logic is a data base dependent ( JDBC persistence logic Database dependent ).
- ④ In JDBC technology, we have all checked Exceptions, whenever we add JDBC code in our application, we need to add either try & catch blocks or throws. It increases the burden on programmers (checked Exceptions)
- ⑤ In JDBC technology some repetitive statements are required to perform any operational database. This repetitive statement is called Boiler-plate code. Bcoz of this code a burden on a programmer will be increased. (Boiler-plate code)
- ⑥ In JDBC, a programmer has to take care about connection management explicitly. It means the program has to handle the connection opening & connection closing with a DataBase. If it is not handled properly then at a later point of time, an application can run out of connections (connection management)
- In order to overcome the above drawbacks of JDBC technology the third party vendors started providing a special kind of SQL ORM software or tool.
- While using ORM SQL, it internally uses again JDBC technology for connecting with database. It means, if ORM SQLs, we are indirectly using JDBC to talk with DataBases.
- The major benefits of ORM SQL's includes Data transfer in objects of the persistence logic independent of databases.

Q:- What is ORM?

Ans:- ORM is a new kind of persistence API, which acts like a bridge b/w

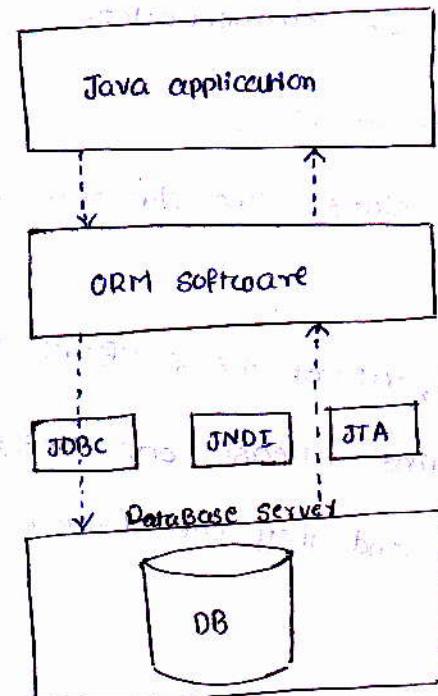
Java application & database and transfers to or from data in the form of objects



06/02/2013

→ ORM softwares internally uses JDBC, JNDI and JTA (Java Transaction API) For communicating a Java application with a DataBase in order to Transfer Objects b/w them.

→ an ORM API uses JDBC Technology to perform Database operations by connecting with it  
→ an ORM API uses JNDI when connection pooling is applied for obtaining Database connections, and JTA is used while managing the Database transactions



The following are the some of the ORM API's given by different vendors

- Hibernate → JBoss (soft tree)
- iBATIS → Apache
- TopLink → Oracle Corp
- OJB (Object Java Bean) → Apache
- etc... → JDO (Java Data object) → Adobe

→ In Real time Applications, the two ORM slw's preferred  
Hibernate, iBATIS.

→ Hibernate is the most ORM popular slw when compared with the other ORM slw's

\*\*\* The primary functionality of all the ORM slw's is same. It means the Data transfer will be done in the form of Objects and the persistence Logic is independent of all the Databases.

Q:- What is Hibernate?

Ans:- Hibernate is an openSource Frame work, also called as an ORM tool. and It acts as a middle layer between a Java appn and DataBase & provides Persistence and query service in the form of Objects.

In other terminology, we call Hibernate as an abstraction Layer on top of JDBC technology.

Hibernate can be called as a tool and also as a framework. Bcz Hibernate follows the commonalities of frameworks.

Frame works follows the following commonalities

- ① Frame works are non-installable slw's. The slw of a Framework will be in the form of a set of jar files.
- ② Frame works are implementations, but not the specifications.
- ③ Every Frame work application contains atleast one configuration file
- ④ Frame works contains jar files and they are divided as main and dependent jar files.

Benefits / Advantages of hibernate :-

- ① Hibernate has an Exception Translator, which converts checked Exceptions of JDBC into unchecked Exceptions of Hibernate. So, Hibernate has all unchecked Exceptions. So, The burden on a hibernate programmer has a reduced, because

a programmer is no need to handle the Exceptions explicitly.

- ② Hibernate has its own query Language called Hibernate Query Language (HQL). With this HQL, Hibernate queries becomes Database independent.
- ③ Hibernate has caching mechanism. It reduces the no.of round trips between a Java application and a DataBase. So, hibernate increases the performance of an application.
- ④ Hibernate has versioning and Timestamp feature. With this feature, we can come to know how many no.of times a data is modified and also on what date at what time the data is last modified.
- ⑤ Hibernate supports Inheritance and polymorphism.
- ⑥ With hibernate we can manage the data stored across multiple tables, by applying Relationships. (associations)
- ⑦ Hibernate is a light-weight framework, because hibernate uses POJO classes objects for DataTransfer b/w an application and a Database.  
etc.....

07/02/2013

### POJO class :-

→ A Java class, which does not extend a base class or does not implement an interface, given by any technology or any framework of Java then such type of Java class is called a POJO class.

→ For example,

```
① public class Test  
{  
    //methods  
}
```

→ here Test is a POJO class. A Java class by default extends java.lang.Object but Object class is a CoreAPI of Java. So, A POJO class can extend on CoreAPI class of Java.

② public class MyException extends Exception

```
{ }
```

//methods

```
}
```

→ here MyException class is a POJO class. Because the Base class Exception belongs to CoreAPI of Java.

③ class Demo

```
{ }
```

```
}
```

class Test extends Demo

```
{ }
```

```
}
```

→ here Test is a POJO class. Because the base class Demo is also a POJO class

④ public class Test implements Serializable

```
{ }
```

```
}
```

→ here Test is a POJO class. Because it is implementing Serializable(I) and this interface is belongs to CoreAPI of Java.

⑤ public class Test extends HttpServlet

```
{ }
```

```
}
```

→ here Test is not a POJO class. Because it is extending HttpServlet and HttpServlet is a class of ServletAPI / ServletTechnology.

⑥ public class Test implements Remote

```
{ }
```

```
}
```

→ here Test is not a POJO class. bcz it is implementing Remote(I) and this Remote(I) belongs to Rmi technology.

→ In Java, a light-weight framework is also called as non-invasive or non-intrusive framework.

→ An Heavy-weight framework is called invasive or intrusive framework.

Q:- What is a JavaBean?

Ans:- A JavaBean is also a Java class, which is independent of any technology or a framework.

→ To say that a Java class as a JavaBean the following two conditions must be satisfied

① A Java class must be "public"

② A Java class must contain a public default constructor.

→ Every JavaBean class is also a POJO class. But every POJO class may or may not be a JavaBean class.

Eg:-

```
public class Demo
```

```
{
```

```
}
```

```
//methods
```

```
}
```

→ Here Demo is a JavaBean class and also a POJO class.

Eg:-

```
public class Demo
```

```
{
```

```
    public Demo(int a, int b)
```

```
{
```

```
}
```

```
//methods
```

```
}
```

→ Here Demo is not a JavaBean class, Demo is a POJO class. Because there is no public default constructor in that class.

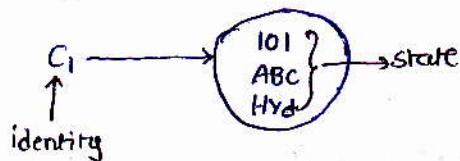
## mapping in Hibernate :-

- mapping is a mechanism of informing an ORM tool (Hibernate) about what POJO class object is need to be stored in what table of a Database.
- mapping is a process of writing mappings between a POJO class name to a table name and POJO class variables names to a table columns name.
- mapping is a metadata, but not the data. Through mapping metadata, Hibernate stores or reads the data using a database.
- Every ORM tool need mapping information, in order to store the state of an object in a DataBase.
- An Object has '3' elements called Identity, state, behaviour.
- In ORM, Storing an Object is nothing but storing the state of an Object, but not the identity & behaviour.

→ For Eg:-

```
public class Customer
{
    private int CustomerId;
    private String customerName;
    private String customerAddress;
    → setters & getters
}
```

Customer c<sub>1</sub> = new Customer();



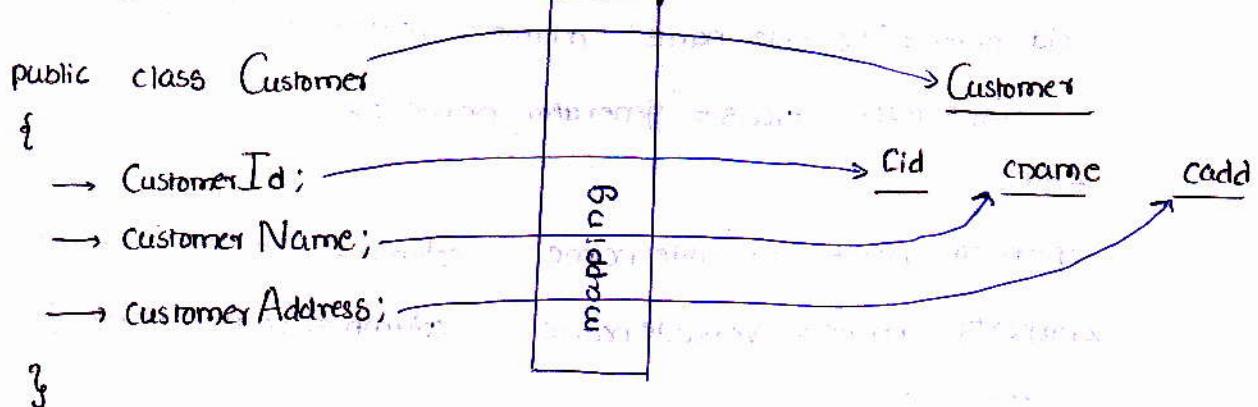
setXxx()  
getXxx()

→ behaviour

mapping is mandatory for an ORM tool, because an application ~~can~~ contain multiple pojo classes and DataBase can contain multiple tables,

so an ORM tool can't understand what class object is need to be stored in what table. To give this information, a programmer must construct / create mapping.

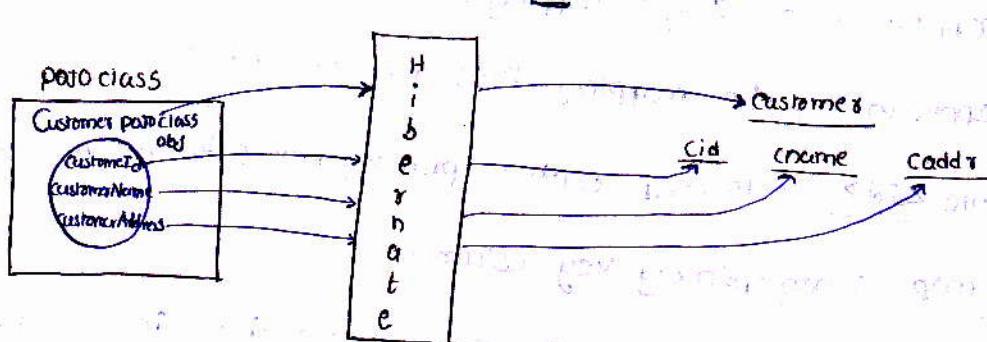
→ A mapping between a Customer class to Customer table will be like the following.



→ in Hibernate, mapping can be done in 2 ways.

① By creating an XML file

② By using Annotations



→ In the above table Customer POJO class object is need to be stored in Customer table in Database.

That POJO class object state will be stored in Customer table columns in DataBase.

→ In hibernate, a mapping file is identified with <classname>.hbm.xml

→ Actually the extension required for an XML file is ".xml" but hibernate creates us suggested to put the extension of a mapping file as ".hbm.xml", to recognize it easily in an application.

→ for in a mapping XML file a Java class and properties of a class are mapped to a Database table and columns of a DataBase table.

→ The Syntax of writing a mapping file in Hibernate is

```

<anyname>.hbm.xml
<!DOCTYPE .....>
<hibernate-mapping>
  <class name="Fully qualified Java classname" table="database table name">
    <id name="Variable name" column="primary key column name">
      <generator class="generator name"/>
    </id>
    <property name="Variable name" column="Column name"/>
    <property name="Variable name" column="Column name"/>
    .....
    .....
  </class>
</hibernate-mapping>

```

#### Imp statements:-

- ① while constructing a mapping file in hibernate, it is possible to write multiple java classes mapping in a single mapping file.
- ② in Realtime applications, the mapping files are created on module basis.
- ③ in a mapping file <Id> is to map with a primary key column and <property> is a tag to map a non-primary key column.
- ④ If there are multiple primary keys in a Database table then instead of <id> tag we use <Composite-id>

#### Configuration in hibernate:-

→ Configuration in Hibernate is nothing but passing 3 types of information to the Hibernate.

- ① Connection properties
- ② Hibernate properties
- ③ Mapping files

→ Configuration in Hibernate can be done in only one way i.e. by constructing an XML file.

→ A configuration file requires extension as '.xml', but hibernate developers suggested to put the extension as ".cfg.xml", to recognize a configuration of Hibernate in a project easily.

→ we can't replace configuration from XML with annotation. It means in hibernate annotations are used only for avoiding mapping XML, but not configuration XML.

NOTE: \*) Hibernate is not a replacement of JDBC. Hibernate is actually introduced as an alternative for EJB2 style entity beans.

\*) Working with Hibernate is nothing but working with JDBC technology indirectly.

→ Hibernate mappings files are write in 2 ways

① Through XML file

② Through Annotations

→ Hibernate configuration files are write in only .xml file. ~~as it is introduced hibernate 3x~~  
also configuration files are write in properties files. These properties files used in hibernate 2.x  
In hibernate 3.x onwards we are using .xml file only.

→ mapping files are used in Realtime the below.

one mapping file contains one per each module in Realtime. A module can contains no. of Java files. i.e. in one mapping file we can <sup>we can</sup> contain no. of Java files

Eg:- <hibernate-mapping>  
      <class name="...>  
          <id ...>  
          <property ...>  
          <property ...>  
      </class>  
      <class name="...>  
          </class>  
      <class name="...>  
          </class>  
      </class>  
</hibernate-mapping>

→ In JDBC code we are using connection properties of database. If we change one DataBase to another DataBase we must open that JDBC code and modifying & last recompile. If we use Hibernate ~~only~~ If we change oracle DB to mySQL DB we open only configuration file & we no need to open each Java code

<any name>.cfg.xml

```

<!DOCTYPE ----->
<hibernate-configuration>
  <session-factory>
    <!-- Connection Properties -->
    <property name="hibernate.connection.driver_class"> fully qualified class name <!-- driver class name --> <!-- property -->
    <property name="hibernate.connection.url"> driver url <!-- property -->
    <property name="hibernate.connection.username"> username <!-- property -->
    <property name="hibernate.connection.password"> password <!-- property -->

    <!-- hibernate properties -->
    <property name="hibernate.dialect"> fully qualified dialect classname <!-- property -->
    <property name="hibernate.show-sql"> true <!-- property -->

    <!-- mapping files -->
    <mapping resource="mapping file name"/>
  </session-factory>
</hibernate-configuration>

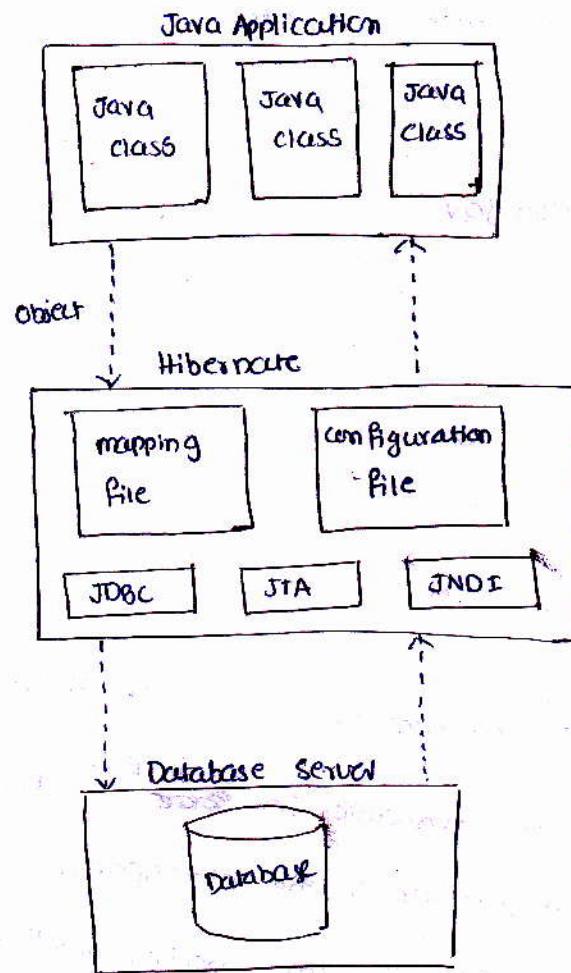
```

Q:- How many Configuration files are required in an Hibernate application?

A:- It depends on the no. of Databases used in an application.  
 → The ratio between the Configuration files and Databases is 1:1.  
 → While working with Hibernate, we must ~~confuse~~ create one configuration file for each Database and it does not matter whether the Databases are running in the Single Db server (or) in multiple Database Servers.

→ In JDBC programmer has to write the SQL operations in Java code manually. It is the responsibility of programmer in JDBC.  
 But in Hibernate we no need to write SQL commands in Java code manually. It is the responsibility of hibernate. So, we use Dialect class in Configuration file. This Dialect class will read internal code of hibernate and generating database specific SQL commands for each operation of hibernate.  
 → ~~about~~ that generating SQL commands if we see then we use show-sql property. This default value is 'false'. If Show-SQL value is true then we see the SQL command. otherwise we can't see.

## Hibernate Architecture :-



### Hibernate Software :

Author : Gavin King

Type : open source framework

Version : 4.x (current version) (compatible with JDK 1.6+)

3.x (compatible with JDK 1.5+)

2.x (j2SDK 1.3+)

download : [www.sourceforge.net/projects/hibernate/files/hibernate3](http://www.sourceforge.net/projects/hibernate/files/hibernate3)

Tutorial : [www.tutorialspoint.com/hibernate](http://www.tutorialspoint.com/hibernate)

[www.roseindia.net](http://www.roseindia.net)

[www.javatpoint.com](http://www.javatpoint.com)

Books : *Hibernate in Action*

*Pro Hibernate 3* (Apress publication)

→ Because Hibernate is a framework, the SW is nothing but a set of jar files.

→ working with framework SWs is nothing but using the predefined classes given in the jar files, to develop an application.

→ For a frame work, we have a main jar file and dependent jar files. similarly

for Hibernate hibernate3.jar is the main jar file and the remaining jar files are dependent jar files of it.

→ The following are the list of jar files setting in to the CLASSPATH for executing a Java application with hibernate.

- ① Hibernate3.jar
- ② antlr-version.jar
- ③ commons-collections-version.jar
- ④ dom4j-version.jar
- ⑤ javassist-version.jar
- ⑥ jta-version.jar
- ⑦ slf4j-api-version.jar
- ⑧ hibernate-jpa-version.jar

12/02/2012

understanding main and dependent jar files :-

- If a Java application uses a class available in ~~a jarfile~~ then at the time of compilation we need to set that jar file in the classpath.
- If a class of one jarfile depends on a class on another jarfile then we need to set the dependent jarfile also in the CLASSPATH

Eg:- public class A  
{  
    void m1()  
    {  
        // code  
    }  
}

>javac A.java  
>jar cvf a.jar A.class

public class B  
{  
    A aobj=new A();  
    void m2()  
    {  
        aobj.m1();  
    }  
}

>javac B.java ↴  
Exception occurs  
>set classpath=a.jar;.;  
>javac B.java ↴  
>jar cvf b.jar B.class ↴

public class Main  
{  
    public static void main(String args[])  
    {  
        B bobj=new B();  
        bobj.m2();  
    }  
}

>javac Main.java ↴  
Exception occurs  
>set classpath=b.jar;.;  
>javac Main.java ↴  
Compiled successfully  
> Java Main ↴

- In the above example code, b.jar is the main jar file and a.jar is the dependent jar file for main.
- While compiling the main class b.jar is set in the CLASSPATH and while executing the Main class along with b.jar, we also need a.jar in the CLASSPATH

### Steps to write a client application in hibernate :-

#### Step-I:-

- Create an object of Configuration class
- Configuration class is said to be the Bootstrapping class of hibernate.
- In a Java application, hibernate environment begins by creating an object of Configuration class.
- Configuration is a class of org.hibernate.cfg package
- After creating an object of Configuration class, we need to call configure() method by passing configuration file as a parameter.
- In this step, along with configuration file mapping files are also loaded and internally an XML parser is used and the data is read from both Configuration and mapping files and stored in some variables of Configuration class.

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

#### Step-II:-

- Built an high level object of hibernate called SessionFactory.
- SessionFactory is an interface of org.hibernate
- The implementation class of SessionFactory interface is SessionFactoryImpl.
- Since hibernate is a framework, both interface and its implementation class are provided by hibernate only.
- When a SessionFactory object is build then all Configuration and mapping will be stored into SessionFactory object.

- SessionFactory Object is called <sup>a</sup>heavy weight Object.
- This is the only one heavy weight Object created in hibernate.

Syntax:-

```
SessionFactory factory = conf.buildSessionFactory();
```

Step-III:- Open a Session of Hibernate.

- When a Session is opened then internally a connection with the DataBase will be established.
- Session is an interface of org.hibernate package. Its implementation class name is SessionImpl.
- in Hibernate sessions are produced by SessionFactory. It means SessionFactory is a factory is producing Sessions(Objects)

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

Step-IV:- Begin a Transaction with a DataBase

- in hibernate all DataBase operations must be done within a Transaction, except "select" operation
- A Transaction can be started using session Object.
- Transaction is an Interface of org.hibernate package. Its implementation class is TransactionImpl.

```
Transaction tx = session.beginTransaction();
```

Step-V:- perform operations on DataBase using methods of session

Step-VI:- Commit a Transaction after completion of operations.

- in Hibernate, all operations must be done within a Transaction.
- If any operation is failed then hibernate automatically rollbacks the transaction.

```
tx.commit();
```

### Step-VII :-

→ close a Session with a DataBase

→ when a Session is closed then internally a Connection with the DataBase can be closed.

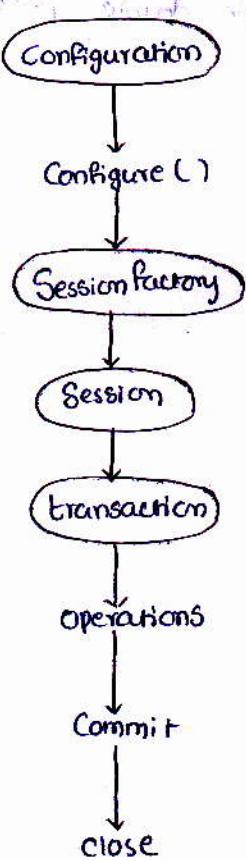
```
Session.close();
```

### Step-VIII :-

→ Close SessionFactory of hibernate

```
factory.close();
```

flow:-



while Storing or search for getter methods for reading it search for setter methods.

The class should have default constructor. Then it stores the value but not read the data. So, default constructor is mandatory.

The following Example is for Storing a POJO class object called product in a database using hibernate.

HbAPP1

product.java → To store the data in POJO class object

product.hbm.xml

hibernate.cfg.xml

productInsert.java

## // Product.java (POJO)

```
public class Product
{
    private int productId;
    private String productName;
    private double price;

    // default constructor
    public Product()
    {
    }
}
```

### // Parameterized Constructor

```
public Product(int productId, String productName, double price)
{
    this.productId = productId;
    this.productName = productName;
    this.price = price;
}
```

### // setters and getters

```
public void setProductId(int productId)
```

```
{
    this.productId = productId;
}
```

```
public int getProductId()
```

```
{
    return productId;
}
```

```
public void setProductName(String productName)
```

```
{
    this.productName = productName;
}
```

```
public String getProductName()
```

```
{
    return productName;
}
```

```
public void setPrice(double price)
```

```
{
    this.price = price;
}
```

```

public double getPrice()
{
    return price;
}
}

```

- In a pojo class of Hibernate application while storing an object in a DataBase, Hibernate checks for getter methods in a pojo class and while reading an object from a DataBase, Hibernate checks for a default constructor and also setter methods in a pojo class. So, to perform either a store or read operations, in a pojo class of Hibernate we should include a <sup>public</sup> default constructor and also setters and getters methods of properties.
- If we donot create any constructor in the Java class then during compilation a public default constructor will be automatically added for that java class.

```

<!-- product.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="Product" table="product-table">
        <id name="productId" column="pid">
            <generator class="assigned"/>
        </id>
        <property name="productName" column="pname"/>
        <property name="price" column="price"/>
    </class>
</hibernate-mapping>

<!-- hibernate.cfg.xml -->
<!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>
        <!-- Connection properties -->
        <property name="hibernate.connection.driver_class">oracle.jdbc.OracleDriver</property>
        <property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:orcl</property>
    
```

```

<property name="hibernate.connection.username"> Scott <property>
<property name="hibernate.connection.password"> Tiger <property>

    <!-- hibernate properties -->
        dialect, OracleDialect
    <property name="hibernate.dialect"> org.hibernate.dialect </property>
    <property name="hibernate.show-sql"> true </property>

    <!-- mapping file properties -->
        resource
    <property name="product-hbm.xml"/>

```

*(mapping file properties) has two types of configurations are present  
 mapping resource*

<session-factory>

<!-- session-factory configuration through a session factory -->

<!-- hibernate-configuration >

*(configuration resource contains all the information and there is no need to have both of them together because both are same)*

*Session factory configuration*

*Hibernate SessionFactory*

<!-- hibernate-configuration >

<!-- dialect - mapping = dialect "Oracle" = ochen EASY </configuration>

<!-- big = small "big" = small "big" = small big </configuration>

<!-- bombing = number "bombing" = number </configuration>

<-- hib

<!-- bombing = number "smallBombing" = small </configuration>

<!-- ring = number "ring" = small </configuration>

<-- hard

<!-- configuration >

<!-- mapping - standard >

<-- mapping - standard >

<-- mapping - standard >

*(mapping file configurations are stored in the XML file and there is no need to have both of them together because both are same)*

// ProductInsert.java

import org.hibernate.\*;

import org.hibernate.cfg.\*;

public class ProductInsert

{

public static void main (String[] args)

{

// Step-I

Configuration conf = new Configuration();

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

// Step-II

SessionFactory factory = conf.buildSessionFactory();

// Step-III

Session session = factory.openSession();

// Create a Product Object

Product p=new Product();

// Set the values

p.setProductId(111);

p.setProductName("Sony");

p.setPrice(7000);

// Step-IV

Transaction tx = session.beginTransaction();

// Step-V

```

        session.save(p);
    // Step-VI
    tx.commit();
    // Step-VII
    session.close();
System.out.println("A Product object is stored in Database");
    // Step-VIII
    factory.close();
}
}

```

→ Product.java is a POJO class, so it can be compiled by without any CLASSPATH setting.

HibernateApp> javac Product.java

→ ProductInsert.java used Hibernate API. So, to compile this Java file, we need to set main jar file (hibernate3.jar) of Hibernate in the CLASSPATH

HibernateApp1> set classpath = D:\hibernate-3.6.5\finaldistribution\hibernate3.jar;

HibernateApp1> javac ProductInsert.java successfully compiled

→ To execute ProductInsert application, we need to set main jar file and dependent jar files (ojdbc14.jar) in the CLASSPATH.

HibernateApp1> set classpath = D:\hibernate-Final distribution - 3.6.5\hibernate3.jar ; D:\hibernate-Final distribution - 3.6.5\lib\required\antlr-2.7.6.jar ; D:\Final distribution - 3.6.5\lib\required\commons-collections-3.1.jar ; D:\H-d\required\lib\dom4j-1.6.1.jar ; D:\H-d\lib\gravassist-3.12.0.jar ; D:\H-d\required\lib\jta-1.1.jar ; D:\H-d\lib\required\stf4j-api-1.6.1.jar ; D:\H-d\lib\JPA\hibernate-jpa-2.0-api-1.0.0.Final.jar ; D:\Oracle\ora92\jdbc\lib\ojdbc14.jar;

→ Before we run this ProductInsert client application, we need to create a table called Product\_table in the DataBase.

Create table product\_table (pid number(5) primary key, pname varchar(15), price number(9,2));  
Table created

→ HibernateApp1> java ProductInsert

→ This insert query is created by internally Dialect class. If we see this command show.sql is recommended.

Hibernate : insert into Product\_table (pname, price, pid) values (?, ?, ?)

↳ Prepared Statement

A Product Object is stored in DataBase.

NOTE:- Hibernate internally uses Prepared Statement of JDBC, to perform DataBase operations.

NOTE:-

- ① While creating a mapping file in Hibernate, if a pojo class variable name and Database table column name are matched then column attribute is optional.
- ② If Pojo class name and Table name are matched then table attribute is optional.

hibernate.hbm2ddl.auto property :-

15/02/2013

- This is an Hibernate property, used for performing DDL operations on a database by hibernate.
- While working with hibernate, hibernate has the ability to automatically execute / perform DDL operations, if necessary.
- The 4 values of this property are
- ① validate
  - ② create
  - ③ update
  - ④ create-drop
- The default value of this property is "validate".
- (i) If the value of this property is validate then hibernate only checks for the table and for columns in the table.
- (ii) If either table or columns does not exist then hibernate throws an Exception.
- If hibernate.hbm2ddl.auto property is "create"
- (i) Then hibernate checks for the table in DataBase. If exist then it will be dropped and then a new table will be created by hibernate.
- (ii) If table doesn't exist then directly a new table is created.

(iii) the Create value is suitable when executing <sup>an</sup> application for first time. while executing for next time, we will lose the previous data added to the table.

→ if hibernate.hbm2ddl.auto property is update then hibernate checks for the table and for columns. if table exist then it checks for columns. if a column does not exist then it will be added to the table.

(i) In case of update value, hibernate does not drop the existing table and does not drop the existing columns. If a table does not exist then hibernate creates a new table in the database.

(ii) Comparatively update is the best value then ~~rescuing~~ values.

→ if hibernate.hbm2ddl.auto property is create-drop then (i) hibernate creates the table when an application execution starts, and drops the table when SessionFactory is closed explicitly in an Application.

(ii) if SessionFactory is not closed explicitly, i.e. if we do not write SessionFactory.close() then hibernate will not drop the table from the DataBase.

→ This hibernate.hbm2ddl.auto property can be added in a Configuration file like the following

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

Q:- Why we prefer to use Configuration filename as "hibernate.cfg.xml"?

A:- If a Configuration filename is hibernate.cfg.xml then it is not mandatory to pass the filename as a parameter for configure() method.

→ By default configure() loads hibernate.cfg.xml file

→ If the configuration filename is Some other name then ~~we have to~~ It must be passed as parameter for configure() method.

→ If filename is hibernate.cfg.xml then conf.configure(); ✓

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

→ If filename is Saithya.cfg.xml then conf.configure(); X

conf.configure("Saithya.cfg.xml"); ✓

Q:- What is the difference b/w hibernate properties file and hibernate configuration file?

Ans:- In hibernate properties file, we can write Connection properties and hibernate properties. But mapping files can't be added to a properties file. But if it is a configuration XML file then we can add hibernate properties, connection properties and also mapping files.

- Hibernate Configuration XML is introduced in hibernate 3.x  
in hibernate 2.x, instead of XML file a properties file is used.
- If properties file is used then the filename should be ""hibernate.properties"" and in a client application it will be loaded automatically, whenever an object of Configuration class is created.

```
Configuration conf = new Configuration();
```

- It means we no need to call configure() method
- In case of properties files, hibernate mapping files are added to the Configuration object by calling a method conf.add(-)

```
conf.addFile("product.hbm.xml");
```

Q:- If hibernate properties file and a Configuration XML file, both are created then which is considered by hibernate?

Ans:- If configure() method is not called then hibernate uses properties file. And if configure() method is called then Configuration XML file.

Q:- Can we create a hibernate application, by without creating hibernate properties file or configuration XML file or not?

Ans:- Yes,  
we need to add the connection properties, hibernate properties and mapping files directly to the Configuration object in the source code.

- The draw back of creating an hibernate application without properties file

or Configuration XML file is, if any changes are required in the properties then we need to make the changes in source code, then we ~~will~~ need to recompile the source code.

If it is a serverside application then we need to reload the application into the server, and sometimes we need to restart the server.

for Eg:-

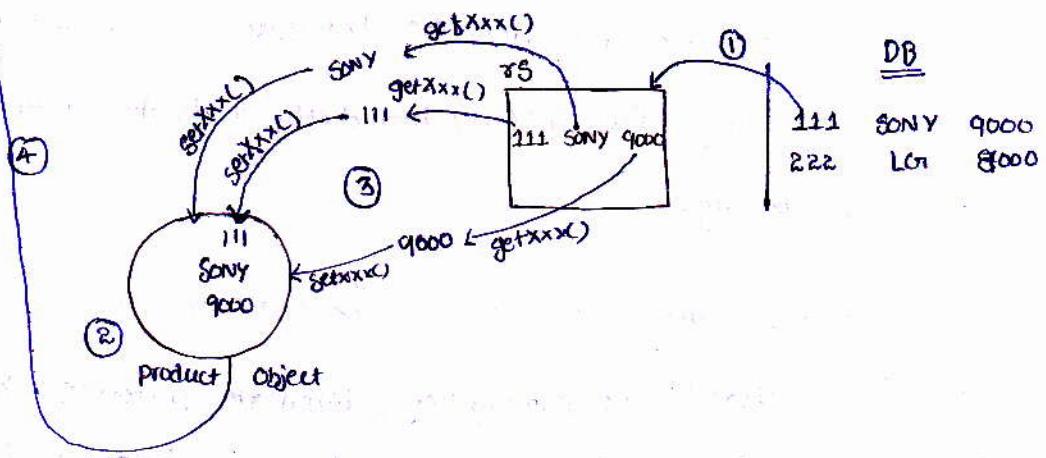
```
Configuration conf = new Configuration();
conf.setProperty("hibernate.connection.driver-class", "oracle.jdbc.OracleDriver");
conf.setProperty("hibernate.dialect", "org.hibernate.dialect.OracleDialect");
conf.setProperty("hibernate.username", "scott");
conf.setProperty("hibernate.password", "tiger");
conf.setProperty("hibernate.show-sql", "true");
conf.setProperty("hibernate.hbm2ddl.auto", "update");
conf.addFile("product.hbm.xml");
SessionFactory factory = conf.buildSessionFactory();
```

## \* load operation (Select Operation) :-

- load operation is nothing but reading an object from a DataBase into a Java app using Hibernate.
- In order to perform a load operation / select operation a Java programmer has to call any one of the following two methods given by Session Interface
  - ① `get(Class clazz, Serializable id)`
  - ② `load(Class clazz, Serializable id)`
- While performing Select operation using hibernate, hibernate need a Class object of a POJO class and ID for identifying one object among multiple objects saved in a Database.
- While loading, hibernate creates internally an object of a POJO class for storing the selected data from a DataBase. For creating an object of the POJO class, hibernate needs Class object of the POJO class.
- hibernate internally calls `newInstance()` for creating an object of a POJO class, using Class object.
- When `newInstance()` is called then it will check for public default constructor in a POJO class. If exist then an object of the POJO class is created. otherwise an exception will be raised. So, while creating a POJO class in hibernate application, a public default constructor is mandatory in the class. Otherwise hibernate fails to load / select the objects from the DataBase.
- While loading / selecting an object from a DataBase then hibernate performs the following ~~steps~~ operations internally
  - ① hibernate first reads data from a DataBase and stores it in a ResultSet object.
  - ② hibernate creates a POJO class object by calling `newInstance()` method
  - ③ hibernate reads the ~~row~~ values from ResultSet, by calling `getXXX()` methods and stores the values into <sup>POJO class object</sup> ~~setXXX()~~ POJO class object by calling `setXXX()` method of POJO class.

④ Finally, hibernate returns the POJO class object to Java appn.

Eg:- Object o = session.get(Product.class, 111);



get(-)

→ When ~~getXXXX~~ method is used for loading an object from database. hibernate  
1<sup>st</sup> verifies weather "id" the given "id" is exist in DataBase or not. If exist  
then immidiately reads it from the DataBase and prepares internally POJO class  
object (Product class obj) and finally returns that POJO class object  
back to our Java appn

→ If the given id is not exist in the DataBase then hibernate skip the internal process  
and finally returns null value back to the Java appn

→ in case of load(-) method, while loading an object, hibernate checks for given id  
and it is exist in the DataBase then hibernate prepares a POJO class object  
with empty 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. so, we call it as a "proxy" object.

→ When a Java program starts accessing that object then internally hibernate will  
reads the data from DataBase and puts it into the POJO class object  
This mechanism in hibernate is called "Lazy loading".

→ if the given ID does not exist in the DataBase then hibernate throws  
ObjectNotFoundException; in case of load method.

Q:- What is the difference b/w a load(-,-) and get(-,-)?

Ans:- ① If the given Id does not exist in the DataBase then load(-,-) method throws an Exception, but get(-,-) returns null.

② load(-,-) loads the data from the DataBase on demand (at the time of accessing).

It is Lazyloading. But, get(-,-) immediately loads the data from a DataBase so, It is early loading.

Q:- Is it mandatory to have a primarykey in the DataBase table, while working with in Hibernate?

Ans:- NO, If there is no primary key, hibernate performs save operation without errors, but in load operation hibernate throws an exception if the given id is found for more than once.

NOTE:- In a mapping file, if the table does not have a primary key then any column can be mapped as "id". It means "id" in a mapping file is mandatory.

// ProductSelect.java

```

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

public class ProductSelect
{
    public static void main(String[] args)
    {
        Configuration cof = new Configuration();
        cof.configure();
        SessionFactory factory = cof.buildSessionFactory();
        Session session = factory.openSession();
        Object o = session.get(Product.class, 333);
        if(o==null)
        {
            System.out.println("A Product with id 333 is not found");
        }
        else
        {
            Product p=(Product)o;
            System.out.println("A Product with id 333 is found");
            System.out.println("product id=" + p.getId());
            System.out.println("Product name=" + p.getName());
            System.out.println("product price=" + p.getPrice());
        }
        session.close();
        factory.close();
    }
}

```

} // main()

} // class

&gt; javac ProductSelect.java

&gt; java ProductSelect

Hibernate: select product0\_.pid from product0 product0 where product0\_.pid = ?

A product with id 333 is found

product id = 333

Product name = Len

Product price = 6000.0

Q:- How to Identify whether an Object is early (or) Lazy loaded from the Databases?

Ans:- In Early loading, Hibernate generates a Select operation, immediately when an Object is Select (or) loaded. It means even though we are not accessing (reading the data) that Object in the code but it is loaded from (selected from) the DataBase.

In Lazy loading, Hibernate generates a Select operation, only when accessing that Object in the code. otherwise a Select operation is not generated by the hibernate.

#### \* \* \* Session-level cache (level 1 cache)

- When a session is opened in Hibernate then internally a Cache is opened (buffer) along with the Session.
- During the session of hibernate, the operations done on the objects will be stored in the cache (buffer) after session. It means if we perform an Insert (or) Update (or) delete (or) Select then the result of this operations will be stored by hibernate internally in the cache of that Session.
- with the help of the cache hibernate reduces the round trips between a Java appn and a DataBase. so that the performance of an Application will be increased.
- For example, if we load an Object from a Database then first hibernate verifies whether this Object is there in the Session cache or not. If Yes, then it reads the object from cache, instead of from DataBase. So, the trips between Java appn and database will be reduced.
- In hibernate, for every session opened (or) generated by SessionFactory contains its own cache.
- As a hibernate programmer, we don't have any settings, either to enable (or) create the cache

or disable (remove) the cache.

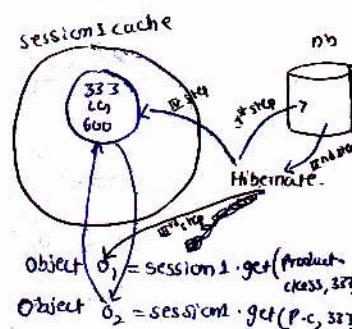
- This session level cache will be automatically created when a session is opened and automatically closed when a session is closed.

19/02/2013

- As a programmer, if you want to remove an object from a Session Cache, by calling evict() method, then we need to pass an object as a parameter for the evict() method.
- While calling clear() method for clearing objects from a Session Cache, no parameter is required for the method, because it removes all objects from cache, not a particular object.

- \* The following code is for loading same object from the DataBase for twice, whose product ID is "333".

```
Object o1 = session1.get(Product.class, 333);
Product p1 = (Product)o1;
S.O.P("Product Name=" + p1.getProductName());
S.O.P("-----");
Object o2 = session1.get(Product.class, 333);
Product p2 = (Product)o2;
S.O.P("Price=" + p2.getPrice());
```



In the above code, an object is loaded within the session for 2 times, but it is loaded for once from the Database and next time it is loaded from the session cache. So, only one select operation will be displayed on console by hibernate.

- In the middle, if we add `Session1.clear()` then an object will be loaded from the DataBase for 2nd time. So, that two select operations will be showed on the console by hibernate.

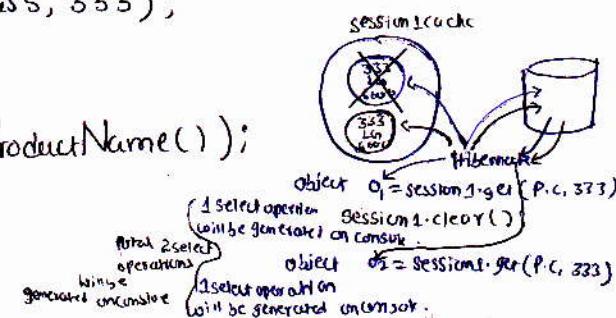
```
Object o1 = session1.get(Product.class, 333);
```

```
Product p1 = (Product)o1;
```

```
S.O.P("Product Name=" + p1.getProductName());
```

```
S.O.P("-----");
```

```
** Session1.clear();
```



```
Object o2 = session1.get(Product.class, 333);
```

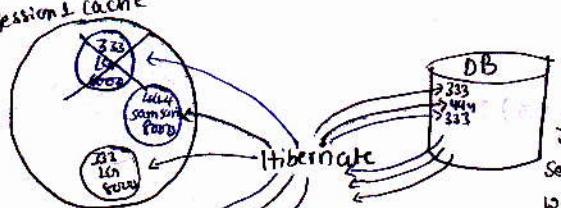
```
Product p2 = (Product) o2;
```

```
S.O.P("Price = " + p2.getPrice());
```

→ In the middle if we want to remove any particular object we use `session.evict()` method. In this `evict()` we pass one object as parameter. That particular object will be removed from the session cache.

```
Object o1 = session.get(Product.class, 333);  
Product p1 = (Product) o1;  
S.O.P("Product name = " + p1.getProductName());  
Object o2 = session.get(Product.class, 444);  
Product p2 = (Product) o2;  
S.O.P("Price = " + p2.getPrice());  
Product p1 = (Product) o1;
```

session cache



```
Object o1 = session1.get(Product.class, 333);
```

```
Object o2 = session1.get(Product.class, 444);  
session1.evict(p1);
```

```
Object o3 = session1.get(Product.class, 333);
```

```
Product p1 = (Product) o1;
```

```
S.O.P("Product name = " + p1.getProductName());
```

```
S.O.P("-----");
```

```
Object o2 = session1.get(Product.class, 444);
```

```
Product p2 = (Product) o2;
```

```
S.O.P("Price = " + p2.getPrice());
```

```
S.O.P("-----");
```

```
session1.evict(p1);
```

```
Object o3 = session1.get(Product.class, 333);
```

```
Product p3 = (Product) o3;
```

```
S.O.P("Price = " + p3.getPrice());
```

3 times  
Select operation  
will be generated,

## Update Operation :-

→ In hibernate, an object which is already stored in a DataBase can be updated in two approaches.

1) we can update an object, by without loading it from database.  
 means now  
 ↑

2) we can update an object by loading it from database.

### Approach - I :-

→ If we want to update an Object, by without loading it from a DataBase then the following steps are need to be follow.

- (i) create a new object of POJO class.
- (ii) set the previous values to the properties, which will not count to update.
- (iii) set new values to the properties which are want to update.
- (iv) call update() of a session with in a Transaction

Eg:- If we want to update a price of a product whose "id" is 333 then the following code is required.

```
product p=new Product();
p.setProductId(333);
p.setProductName("LG");
p.setPrice(4000);
Transaction tx = session.beginTransaction();
session.update(p);
tx.commit();
```

→ when the above code is executed then in the DataBase the price of the product is updated to 4000, but the product name becomes the same name as "LG".

Product - Table

| Pid | Pname | Price      |
|-----|-------|------------|
| 333 | LG    | 6000 → old |
|     |       | 4000 → new |

→ The drawback of this approach is, we need to set both the values which we do not want to update and the value which we want to update. It is a burden on the programmer and also, if a property is not set then its default value will be updated in the Database.

Eg:- Product p = new Product();

p.setProductId(333);

p.setPrice(8000);

Transaction tx = session.beginTransaction();

session.update(p);

tx.commit();

Product-table

| Pid | Pname | Price                    |
|-----|-------|--------------------------|
| 333 | null  | 6000 → old<br>8000 → new |

product p = new Product();

p.setProductId(333);

p.setName("SAMSUNG");

Transaction tx = session.beginTransaction();

session.update(p);

tx.commit();

Product-table

| Pid | Pname   | Price              |
|-----|---------|--------------------|
| 333 | SAMSUNG | 0 → default value. |

### Approach-II:-

→ In this approach, 1<sup>st</sup> we need to load an Object, which we want to update from the DataBase and then we can set the new values to the properties which we want to update, with in a Transaction.

→ In this approach, we no need to set previous values to the properties which we do not want to update.

→ In this approach, we no need to call update() method of a session

Eg:- If we want to update the price of a product Id "333" then the

following code is required.

Object o = session.get(Product.class, 333);

Product p = (Product) o;

Transaction tx = session.beginTransaction();

p.setPrice(8000);

tx.commit();

in The table

Product-table

| Pid | Pname | Price                    |
|-----|-------|--------------------------|
| 333 | LG    | 6000 → old<br>8000 → new |

- when updating an object while loading it from a database, the modifications done on the data of an object will be stored <sup>1<sup>st</sup></sup> in cache of a session. later, when a Transaction is committed then the changes are transferred the cache memory to the DataBase. This operation in hibernate is called as "flushing"
- Hibernate performs flushing, only when if changes exist ~~ing~~ between the state of an object in cache and the database.
- If there are no changes in the state of an object stored in cache and stored in Database then hibernate does not apply any flush operation when a transaction is committed.
- Flushing is a mechanism of transferring the state changes from a cache memory object to a DataBase.

The following client application of Java is for loading and updating an object, whose productId is 333.

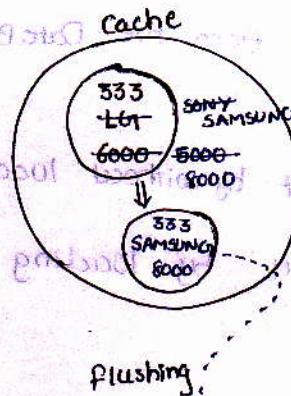
```
// ProductUpdate.java
import org.hibernate.*;
import org.hibernate.cfg.*;
public class ProductUpdate
{
    public static void main(String[] args)
    {
        Configuration conf = new Configuration();
        conf.configure();
        SessionFactory factory = conf.buildSessionFactory();
        Session session = factory.openSession();
        Object o = session.get(Product.class, 333);
        Product p = (Product)o;
        Transaction tx = session.beginTransaction();
        p.setPrice(5000);
        p.setPrice(8000);
    }
}
```

↳ primitive data type (or) wrapped data  
Serialized & Deserialized

```

    p.setProductName("SONY");
    p.setProductName("SAMSUNG");
    tx.commit();
    session.close();
    factory.close();
}

```



product\_table

| pid  | pname   | price |
|------|---------|-------|
| 333  | SONY    | 6000  |
| 333  | TOT     | 6000  |
| 6000 | SAMSUNG | 8000  |

SONY      TOT      6000

SAMSUNG      8000

new object in memory

(\* - Standard class)

(\* - Object class)

new object in memory

(new [Object] in memory)

(Memory free) = new (Memory management)

(Memory) = free

new (Object) = new (Memory)

new (Object) = new (Memory)

(Object) = new (Memory)

new (Object) = new (Memory)

(Object) = new (Memory)

## \* ) Delete Operation :-

→ We can Delete an Object from the DataBase using Hibernate in the following two ways.

- (1) we can delete an object by without loading it from the DataBase.
- (2) we can delete an object by loading it from the DataBase.

### Approach - 1 :-

- In this approach, we can create an object and we can assign the Id to be deleted from the DataBase and then we can call delete() method to delete an object from the DataBase.
- While deleting an object in this approach we no need to assign any other values an object except Id. Because Hibernate deletes an object from database by checking for the Id.
- For example, If we want to delete a product whose Id is 333 from the DataBase, by without loading it from the DataBase then the following code is required.

```
//ProductDelete.java
import org.hibernate.*;
import org.hibernate.cfg.*;

public class ProductDelete
{
    public void main (String[] args)
    {
        Configuration conf = new Configuration();
        conf.configure();
        SessionFactory factory = conf.buildSessionFactory();
        Session session1 = factory.openSession();
        Product p = new Product();
        p.setProductId(333);
        Transaction tx = session1.beginTransaction();
        session1.delete(p);
```

```
tx.commit();
Session1.close();
factory.close();
}
```

### Approach - 2 :-

- In this approach, we need to load an object from the Database by calling `get(-,-)` or `load(-,-)` method and then we need to delete the loaded object by calling `delete()` method.
- When compared with this approach<sup>2</sup>, in approach 1, it is enough to call `delete()` method in approach 2. So, comparatively approach 1 is better than approach 2.

```
Object o = session.get(Product.class, 833);
```

```
Transaction tx = session.beginTransaction();
```

```
session.delete(o);
```

```
tx.commit();
```

## DataBase Operations

### Single row operation

- ① save()
- ② update()
- ③ delete()
- ④ load()

⋮

### Bulk operation

- ① HQL
- ② NativeSQL
- ③ Criteria

## Versioning an Object:-

- In Hibernate, we have a feature that hibernate stores Version of an object in a DataBase table column, to identify how many no.of times an object is updated, since from its creation.
- When an object is newly inserted into the DataBase then its version number is inserted as '0'. When its data is updated then hibernate automatically increments its version by '1', and stores that version number in DataBase.
- By reading the version of an object, we can find that how many no.of times the object is updated so far..
- To get Versioning feature for a POJO class object then the following two changes are required in an application.
  - ① in POJO class, we need to create an Integer variable and we need to generate appropriate setters and getter methods.
  - ② In mapping file, we need to configure a tag called <version> with Version Variable and Version column used in a POJO class and DataBase table respectively.
- This <version> tag must be configured immediately after <id> tag in a mapping file.

→ To apply this Versioning feature of hibernate to the product objects then we need to make the changes in `Product.java` and `Product.hbm.xml` like the following

// `Product.java`

```
public class Product {  
    private int productId;  
    private String productName;  
    private double price;  
    private int v;  
    → setters  
    → getters  
}
```

`<!-- Product.hbm.xml -->`

`<hibernate-mapping>`

```
<class name="Product" table="Product_table">  
    <id name="productId" column="pid">  
        <generator class="assigned"/>  
    </id>  
    <version name="v" column="ver-id"/>  
    <property name="productName" column="pname" length="10"/>  
    <property name="price" column="price"/>  
</class>  
</hibernate-mapping>
```

→ when a new product object is inserted in a Database then

21/02/2013

its version number will be saved as "0". If a product is loaded and updated then its version will be incremented by "1" automatically by hibernate.

product\_table

pid	pname	price	ver_id
111	SONY	9000	0

→ when a new product is inserted its version number is 0  
→ whenever a product is updated its version number will be incremented by 1

### Timestamp Feature of Hibernate:

→ in Hibernate, it is possible to know, on what date at what time an object is updated

using the Timestamp feature of Hibernate.

→ Hibernate automatically stores (or) records the System date and time into the Database, whenever an object is inserted (or) whenever an object is updated.

→ with this Timestamp feature of Hibernate, it is only possible to know when an object is updated in the Database. but it is not possible to know how many no.of times an object is updated.

→ It is possible to know how many no.of times <sup>an</sup> object is updated using Versioning feature of Hibernate. but it is not possible to apply both

Versioning and Timestamp at a time in an appn.

→ To apply Timestamp features of Hibernate then the following changes are required in an Application.

- ① in POJO class, we need to create a property of type Timestamp and generate Settor and geter methods.
- ② in mapping file of hibernate, configure <timestamp> immediately after <id> tag.

→ In POJO class, while creating a property of Timestamp we need to import `java.sql.*;` package, because Timestamp is a class of java.sql package.

Example:-

→ To apply this Timestamp feature for product example, we need to make the changes in `product.java` and `product.hbm.xml` like the following

```
// Product.java (POJO)  
import java.sql.*;  
protected public class Product  
{  
    private int productId;  
    private String productName;  
    private double price;  
    private Timestamp ts;  
    → setters  
    → getters  
}
```

<!-- Product.hbm.xml -->

<hibernate-mapping>

```
<class name="Product" table="product_table">  
    <id name="productId" column="pid">  
        <generator class="assigned"/>  
    </id>  
    <timestamp name="ts" column="tstamp"/>  
    <property name="productName" column="pname" length="10"/>
```

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

</class>

</hibernate-mapping>

→ If Hibernate is creating a DataBase table then Hibernate uses the Datatype of

Timestamp column as Date. If it is Datatype then it can store only on what

date the an Object is updated. but it can't store on what time an object is updated.

→ If we create DataBase table manually then we can use Datatype of a

Timestamp column as timestamp. so that hibernate stores both

Date and time in the DataBase whenever an object is created or updated.

SQL > Create table product\_table( pid number(5) primary key, pname Varchar(10),  
price number(9,2), tstamp timestamp);

Table created.

→ If we run ProductInsert Client application them hibernate stores a  
Product object and also the system date and time like the following

in DataBase.

SQL > Select \* from product\_table;

pid	pname	price	tstamp
111	SONY	8000	21-Feb-13 08:00:40.35900 AM

## Steps to develop an Hibernate application Using MyEclipse 8.x.

Step-I:- create a table called Employee in the DataBase like the following

SQL> create table employee (empno number(5) primary key, ename varchar(20),  
Sal number(8), Deptno number(3));

Table created.

Step-II:- start MyEclipse IDE

Start → programs → MyEclipse → My Eclipse 8.x Stable → Enter  
workSpace [C:\work1] → OK

Step-III:- click on File menu → New → JavaProject → projectname : APPR  
→ Finish

Step-IV:- By default a packageExplorers view is displayed at leftside. In this view  
"bin" folder is invisible. So change to Navigator

↳ stores .class file

In src Folder  
Java source program  
with source

click on Windowo menu → showview → Navigator  
MyEclipse → Add Hibernate capabilities  
Right click on project name → next → next → enter the following details.

DB Driver : [ ]

Connect URL : JDBC : oracle : thin : @localhost : 1521 : orcl

Driver class : Oracle.jdbc.OracleDriver

Username : Scott

Password : tiger

Dialect : oracle 9i/10g

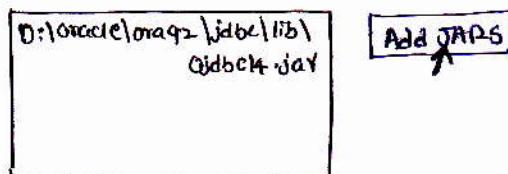
next → deselected  create sessionFactory class

→ Finish

Step-VI:- click on Windowo menu → Open perspective → MyEclipse DataBase Explorer  
→ at left side in a DB Browser view, click on  ViewMenu button → New  
→ enter the following Details.

Driver Template : Oracle (Thin Driver)  
 Driver name : MyDriver  
 Connection URL : jdbc:oracle:thin:@localhost:1521:ord  
 Username : Scott  
 Password : tiger

### DriverJARs



Driver class name: Oracle.jdbc.driver.OracleDriver

→ next → Finish

Step-VII :- in DB Browser View at left side, Right click on MyDriver → open a connection

→ Enter password: tiger → OK

Step-VIII :- Expand MyDriver → Expand Connected to MyDriver → Expand SCOTT

→ Expand TABLE → select Employee table → Right click on Table name and select Hibernate Reverse Engineering → do the following

Java Src folder: /App2/src  →  Create POJO DB Table mapping information



→  Java Data Object (POJO <=> DB Table) → next → ID Generator assigned

Create abstract class

→ next → Finish

Step-IX :- Right click on project name → new → class → name: StudentInsert

→ Finish

```

import org.hibernate.*;
import org.hibernate.cfg.*;
public class StudentInsert
{
  public static void main(String[] args)
  {
  }
}
  
```

```
Configuration conf=new Configuration();
```

```
conf.configure();
```

```
SessionFactory factory=conf.buildSessionFactory();
```

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

```
Employee e=new Employee();
```

```
e.setEmpno(1121);
```

```
e.setEname("Manoj");
```

```
e.setSal(6000);
```

```
e.setDeptno(10);
```

```
Transaction tx=session.beginTransaction();
```

```
session.save(e);
```

```
tx.commit();
```

```
System.out.println("Employee saved in DataBase");
```

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

```
factory.close();
```

Step - II :- Right click on project name → properties → Java Build Path  
→ Add External JARs Button → select Oddbc14.jar → open → OK

Step - III :- Right click on StudentInsert.java → Run as → Java application

(about 80 < 300) radio box  radio button

defeat ← desire

about 80 < 300 radio box  about using no do it self II - 918

defeat ←

\*.class, \*.java, \*.log

(\*.\*).class, \*.java, \*.log

translatable class files

class file (.class) under bin directory

## Different States of an Object in hibernate : (life cycle of an object)

→ A POJO class object <sup>of</sup> to Hibernate is going to have the following 3 states in an Application.

① Transient state

② Persistent state

③ Detached state

### ① Transient State :-

→ When a new object is created for a POJO class then that object will be in a Transient state.

→ If we create a POJO class object by assigning "null" value then also we say that an object is in Transient state.

→ A Transient State Object is an object which is not yet associated with a Session of Hibernate.

→ A Transient State Object is not representing a row of an underline Database.

→ If we make any changes on the Data of a Transient Object then this changes are not effected on the DataBase.

Eg:- Product p=new Product();  
 p.setProductId(111);  
 p.setProductName("nani");  
 p.setProductPrice(3000);

→ here "P" is in Transient state.

Product p=null;

→ here "P" is in Transient state.

### ② Persistent State :-

→ when a save operation is done on a Transient State Object (or) when a load operation is done on a Transient state Object then the Object will be entered into persistent state.

→ When an Object is entered into persistent state then it is associated with session of Hibernate.

→ If any modifications are done on an object in a persistent state then  
the changes are effected on the underlying Database also.

Eg(1):

```
Product p=new Product();
```

```
p.setProductId(111);
```

```
p.setProductName("ABC");
```

```
p.setPrice(9000);
```

→ here 'p' is in Transient state

```
Transaction tx=session.beginTransaction();
```

```
session.save(p);
```

→ here 'p' is in Persistent state

Eg(2):-

```
Product p=null;
```

→ here 'p' is in Transient state

```
Object o=session.get(Product.class,111);
```

```
p=(Product)o;
```

→ here "p" is in Persistent state.

(3) Detached State:-

→ When a session is closed or when a session is cleared or when an object is evict from the session then an object is converted from persistent state into Detached state.

→ When an object loses its attachment with a session then it will be entered into Detached state.

→ When an object is in Detached state and if <sup>any</sup> modifications are done on that object then the changes are not effected on the Database.

→ The commonality in Transient state and Detached state is, in both cases the changes are not effected on a Database.

→ Detached state is a state when an object is coming out of a session and Transient state is a state before entering into a session.

Eg(1)

```
Product p=new Product();
```

```
p.setProductId(111);
```

```
p.setProductName("ABC");
```

```
p.setPrice(9000);
```

→ here 'p' is in Transient state

Eg:-②

```
Product p=null;
```

→ here "p" is in transient state

```
Object o=session.get(Product.class,111);
```

```
p=(Product)o;
```

→ here "p" is in persistent state

```
session.clear();
```

→ here "p" is in Detached state.

Transaction tx = session.beginTransaction();

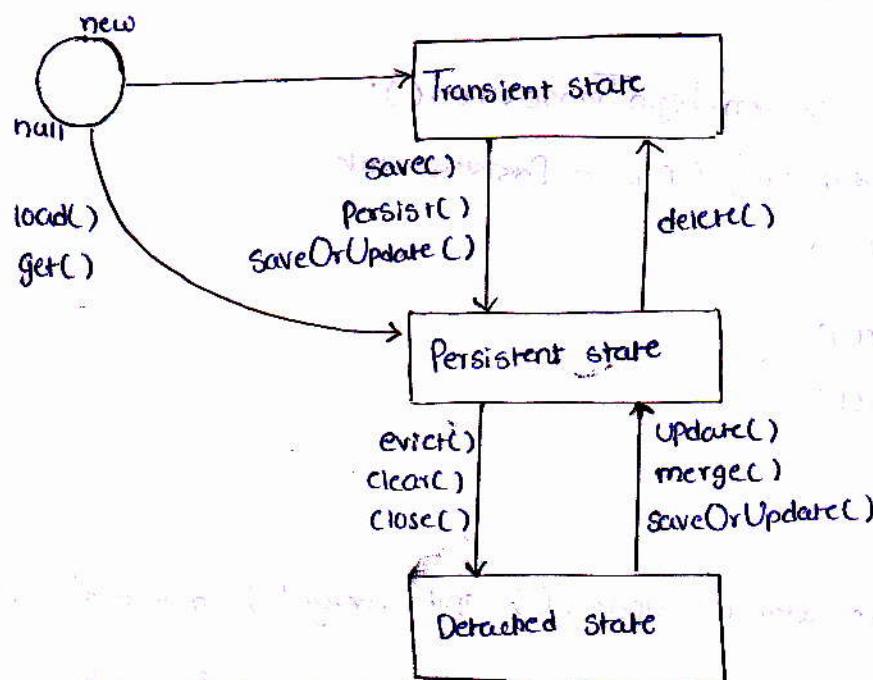
Session.save(p);

→ here 'p' is in persistent state

tx.commit();

session.close();

→ here 'p' is in detached state.



23-02-2015

NOTE:- Transient State is an Object which is not yet attached to a session of hibernate and Persistent State is an Object attached to Session of hibernate and Detached State is an Object which is came out of a Session.

//StatesClient.java

```
import org.hibernate.*;
```

```
import org.hibernate.cfg.*;
```

```
Class StatesClient
```

```
{ public static void main(String args[])
```

```
{
```

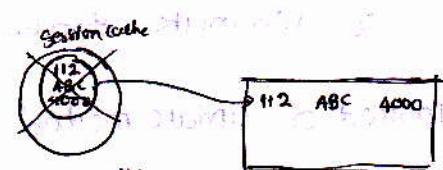
```
SessionFactory factory = new Configuration().configure().buildSessionFactory();
```

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

```
Product p = new Product(); //p is in transient state
```

```
p.setProductId(112);
```

```
p.setProductName("ABC")
```



p.setPrice(4000);

```
Transaction tx = session.beginTransaction();
session.save(p); // p is in Persistent state
tx.commit();

Session.clear(); // p is in Detached state
p.setPrice(3000);
```

```
Transaction tx2 = session.beginTransaction();
session.update(p); // p is in Persistent state
tx2.commit();

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

\*\* Q:- What is the difference between update() and merge() method of hibernate?

Ans:- While Converting<sup>a</sup> detached Object into persistent state, in the cache of ~~the~~ session

if already an Object is saved with the same Id (identifier) value then update()  
method fails to convert an object from detached state into persistent.

So, Hibernate throws org.hibernate.IdNotUniqueObject Exception.

→ Instead of update method if we call merge() then hibernate only copies the  
State changes (Values) of detached state Object into an Object in the cache with  
the Same Identifier value (Id value) and then the changes are updated to  
the DataBase when the Transaction is committed.

→ while Converting a detached Object into persistent then it's always  
better to call merge() instead of update() method.

### // UpdateMerge.java

```
import org.hibernate.*;  
import org.hibernate.cfg.*;  
  
class UpdateMerge  
{  
    public void main(String args[]){  
        SessionFactory factory = new Configuration().configure().buildSessionFactory();  
        Session session1 = factory.openSession();  
        Product p1 = null; // p1 → transient  
        Object o = session1.get(Product.class, 111);  
        p1 = (Product)o; // p1 → persistent  
        session1.close();  
        p1.setPrice(10000); // p1 → detached
```

This modifications are not done in the Database in detached state object. modifications are done in the only persistent state.

```
Session session2 = factory.openSession();
```

```
Product p2 = null; // p2 → transient
```

```
Object o2 = session2.get(Product.class, 111);
```

```
p2 = (Product)o2; // p2 → persistent
```

```
Transaction tx = session2.beginTransaction();
```

```
// session2.update(p1); → throws Exception
```

```
session2.merge(p1); // p1 → persistent
```

```
tx.commit();
```

```
session2.close();
```

```
factory.close();
```

```
}
```

## Composite -id :-

- If a DataBase table has morethan one primary key column then while writing mapping file of hibernate, instead of using `<id>` tag , we need to use `<composite-id>` tag.
- While writing the mapping file of hibernate, it is the mandatory that either `<id>` tag or `<composite-id>` tag must be configured
- When a table has a Single primary key (or) no primary keys then we need to Configure `<id>` tag in mapping file. If morethan one primary keys in table then `<composite-id>` tag is required.

## Syntax:-

```
<composite-id>
  <key-property name="Variable name" column="primary key colname"/>
  <key-property name="Variable name" column="primary key colname"/>
</composite-id>
```

25-02-2013

- While Saving an Object into the DataBase with CompositeId, we need to create a POJO class object, and we assign properties for the object and then we save it in DataBase. It means we dont have any new step while saving an object.
- When loading an Object from DataBase, we need to pass Composite-ID for hibernate to load it from a DataBase. It means we need to create an object of POJO class, then we need to assign Primarykey properties for that object and then we need to pass that object as a parameter for either get (or) load methods to load an object from a DataBase.
- At the time of loading an Object, to pass a POJO class object as a second parameter for either get or load methods, the object must be a Serialized object.
- To make a POJO class object as a Serialized object, ~~we~~ implement the class from `java.io.Serializable`(interface)

### Example :-

- In the following Example, we are going to store and load book class objects in a DataBase. Here bookId & isbnNumber are taken as Primary keys. i.e. Composite id
- At the time of saving an object in DataBase, we do not have any changes but at the time of loading a Book, we are passing bookId & isbnNumber as a book object for get(--) load(--) methods.

### Composite Approach

↳ Book.java  
book.hbm.xml  
hibernate.cfg.xml  
BookInsert.java  
BookSelect.java  
\*.class

//Book.java

public class Book implements java.io.Serializable

{  
    private int bookId;  
    private String bookName;  
    private double price;  
    private int isbnNumber;

Setters & getters

}

//Book.hbm.xml

<!-- DOCTYPE -->  
<hibernate-mapping>

<class name="Book" table="book\_info">

<composite-id>

<key-property name="bookId" column="bid"/>

<key-property name="isbnNumber" column="isbn"/>

</composite-id>

<property name="bookName" column="bname" length="10"/>

<property name="price"/>

</class>

</hibernate-mapping>

NOTE:- While creating a mapping file of hibernate, either <id> or <composite-id> tag is mandatory.

### //BookInsert.java

```
import org.hibernate.*;  
import org.hibernate.cfg.*;  
public class BookInsert  
{  
    public static void main(String[] args)  
    {  
        SessionFactory factory = new Configuration().configure().buildSessionFactory();  
        Session session = factory.openSession();  
        Book b = new Book();  
        b.setBookId(1);  
        b.setBookName("Java");  
        b.setPrice(300.0);  
        b.setIsbnNumber(10111);  
        Transaction tx = session.beginTransaction();  
        session.save(b);  
        tx.commit();  
        session.close();  
        factory.close();  
    }  
}
```

### //BookSelect.java

```
import org.hibernate.*;  
import org.hibernate.cfg.*;  
public class BookSelect  
{  
    public static void main(String[] args)  
    {  
        SessionFactory factory = new Configuration().configure().buildSessionFactory();  
        Session session = factory.openSession();  
    }  
}
```

Book b=new Book();

b.setBookId(1)

b. set ISBN Number (10/11);

Book b1 = (Book) session.get(Bo

```
S.o.p("BookName= "+b1.getBookName());
```

```
g.o.p("price = " + b1.getPrice());
```

Session::close();

```
factory.close());
```

Amidopterin pricing has changed in the Database.

In the following table is required to map each box (10).

→ Before executing this example, the table number(5) → bname VARCHAR2(10),

SQL> Create table book\_info ( bid number(5),  
 price number(4,2), isbn number(5), primary key(bid, isbn));

```
>javac *.java
```

```
>java BookInsert
```

Hibernate: insert into book\_info (book\_id, ...)

```
> java BookSelect
```

Hibernate: select book0\_.bid

Bookname = ~~book~~<sup>java</sup>  
price = 300.0

## Generators in Hibernate

- Before storing an object in a DataBase, hibernate verifies whether an "id" value for that object is set or not.
- In Hibernate Id value for an object can be generated by either a programmer or hibernate. This information will be given to the hibernate using <generator> tag as a subtag of <id> tag.
- In Hibernate, the framework has already provided some predefined primary key generation algorithms. and existing algorithms are not suitable for an application then hibernate has given a procedure for creating user-defined primary key generation algorithms.
- a list of predefined generators given by Hibernate (are) are:
  - ① assigned
  - ② increment
  - ③ sequence
  - ④ Identity
  - ⑤ hilo
  - ⑥ native
  - ⑦ foreign
  - ⑧ uuid.hex
  - etc----

26/02/2013

### ① assigned :-

- The default generator class in Hibernate is "assigned"
- If generator class is assigned then hibernate is expecting the id value for an object from a programmer, before an object is going to be saved in the DataBase.
- In case of assigned generator, it is not mandatory to add generator as the subtag for <id> tag.

→ This assigned generator is DataBase Independent. It means it works (assigned) on any Database.

For Eg:-

```
<id name="productId" column="pid">
  <generator class="assigned"/>
</id>
(Or)
<id name="productId" column="pid"/>
```

### ② increment:-

→ If generator class is added as "increment" then hibernate uses a formula called max(Id) + 1. and generates an "id" value and then assigns this Id value for the new object, before it is going to save an object in DataBase.

→ If a programmer is set the value for the <id> property of an object then hibernate simply ignores the "id" value set by the programmer.

→ If a DataBase table does not contain any rows then Hibernate assumes max(Id) as "0"

→ This increment generator is a DataBase independent generator.

Eg:-

```
<id name="productId" column="pid">
  <generator class="increment"/>
</id>
```

→ At the time of Save operation, Hibernate is going to generate a Select and Insert commands

Hibernate: select max(pid) from product

Hibernate: insert into product(pname, price, pid) values (?, ?, ?)

### ③ sequence:-

→ If generator class is sequence then hibernate selects next value of a sequence from the DataBase and assigns that value as an "id" for the object and then that object will be saved in the DataBase.

- while Configuring this generator class Sequence, we need to pass a parameter called "sequence" with the sequence name created in DataBase as a value.
- If a sequence name is not passed as a parameter then hibernate creates its own sequence called Hibernate-Sequence and selects next value of the sequence and assigns it as an "id," before an Object is going to save in a DataBase.
- This sequence generator is a DataBase dependent. For Example, it works for Oracle only but it does not work for mysql.

→ Eg1:-

```
<id name="productId" column="Pid">
  <generator class="sequence"/>
</id>
```

while save an Object (product Obj) in a DataBase, hibernate is going to generate the following SQL Commands.

Hibernate: select hibernate-sequence.nextval from dual

Hibernate: insert into product(pname, price, pid) values of (?, ?, ?)

→ Eg2:-

```
SQL> create sequence my-sequence increment by 5;
```

```
<id name="productId" column="Pid">
```

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

```
    <param name="sequence"> my-sequence </param>
```

```
</generator>
```

```
</id>
```

while saving an object (product obj) in a DataBase, hibernate is going to generate the following SQL Commands

Hibernate: select my-sequence.nextval from dual

Hibernate: insert into product(pname, price, pid) values of (?, ?, ?)

#### ④ hilo:

- This generator class uses a special formula ~~for~~ generating id for an object, before it is going to store in a DataBase.
- When this generator is executing for 1<sup>st</sup> time then the id value generated is 1.
- When this 'hilo' generator is executed for 2<sup>nd</sup> time onwards then it uses a formula for generating the id value called "max\_10 \* column + column"
- While configuring this hilo generator in a mapping file, along with generator we need to configure 3 parameters
  - ① table
  - ② column
  - ③ max\_10
- If we don't configure the above 3 parameters along with "hilo" generator then hibernate takes default values for the parameters as
  - table → hibernate\_unique\_key
  - column → next\_hi
  - max\_10 → 32767
- When first time hilo generator is executed then after generating the id, hibernate stores a value "1" in next\_hi column of hibernate\_unique\_key table.
- When hilo generator is executed for 2<sup>nd</sup> time then 1<sup>st</sup> it calculates the next id by using the formula and then increments the value of next\_hi <sup>column</sup> by "1". i.e. next\_hi value becomes '2'.

→ If we configure the above '3' parameters then hibernate considers programmer assigned values, instead of the default values.

for Eg:-

```
<id name="productId" column="pid">  
  <generator class="hilo">  
    <param name="table"> my-table </param>  
    <param name="column"> next </param>  
    <param name="max-10"> 10 </param>  
  </generator>  
</id>
```

```
delete from product;  
commit;  
drop table product;  
drop table hibernate_unique_id;  
commit;
```

→ when a product object is saved using this "hilo" generator for 1<sup>st</sup> time then productId will be assigned as "1". After generating the "Id", hibernate stores a value "1" in next column of my\_table.

→ After executing same operation, we will get the following output into the DataBase.

```
SQL> select * from product-table;
```

pid	pname	price
1	sony	8000

```
SQL> select * from my-table;
```

next
1

→ If another product is saved in DataBase then "hilo" generator generates the "id" value as "11" ( $\text{max-10} * \text{next} + \text{next}$ ) (i.e.  $10 * 1 + 1 = 11$ ). After generating the id, hibernate increments the value of next column by 1

→ At DataBase we will get the following output.

```
SQL> select * from product-table;
```

pid	pname	price
1	sony	8000
11	Lor	6000

```
SQL> select * from my-table;
```

next
2

→ This hilo generator is Independent of Databases.

- while configuring this "hilo" generator, we can also configure some parameters instead of all the three in xml file.

### (5) Identity:-

- This Identity generator is used for representing autoincrement column of a DataBase table, in a mapping file of hibernate.
  - autoincrement column is not supported by all DataBase servers. So, this Identity is a DataBase dependent generator.
- for Eg:- oracle doesn't support autoincrement column, but MySQL supports autoincrement column. So, Identity generator works for MySQL, but not for oracle.

```
Eg:- <id name="StudentId" column="sno">
      <generator class="identity"/>
    </id>
```

### (6) native:-

- This generator is not a new generator class. It is equal to any one of the following 3 generators.
  - ① sequence
  - ② identity
  - ③ hilo
- Hibernate 1st checks sequence is supported by the DataBase or not. If yes then native is equal to sequence.
- If not then hibernate checks for Identity. If yes then native is equal to Identity.
- If not then finally hibernate uses "hilo".

```
Eg:- <id name="productId" column="pid">
      <generator class="native"/>
    </id>
```

### (7) Foreign:-

- This generator class is only useful when applying one-one relationship b/w the objects. This Foreign generator is used to copy the id value of one object to Id value of another object.

→ This foreign generator is independent of DataBases.

### uuid.hex:

- This generator class is used for a primary key of "String type" in a DataBase table. UUID stands for Universal Unique id.
- UUID is an algorithm it generates a Unique String of 32 hexadecimal digits, by based on the following 4 values.
- ① IP address of the System.
  - ② Startup time of JVM
  - ③ System time
  - ④ Counter value in JVM
- for String type of primary key columns, either we can use generator class as this UUID.hex (or) assigned.
- The remaining generator classes can be used only for primarykeys of number type.

→ In most of the cases, Primary keys in tables are taken as number type only. So, this uuid.hex is not a frequently used generator class.

## Creating a Custom generator class :-

28/02/2013

- Hibernate framework has provided some predefined generator class is @generating for id's for an objects. If the existing generator classes are not suitable for our application then we can also create an ~~this~~ user-defined generator class we call such generator class as a "Custom generator".
- Creating a Custom generator in an Hibernate appn involves two types.
- Step-I:- We need to create a generator class by implementing it from an interface org.hibernate.id.IdentifierGenerator
- Step-II:- Apply the generator class for an <id> tag in a mapping file using <generator> tag
- IdentifierGenerator is an interface provided a single abstract method called generate(). so, we need to implement generate() with the logic, to generate an Unique Identifier for an Object.

Eg:- public class MyGenerator implements IdentifierGenerator  
{  
    public Serializable generate(SessionImplementor si, Object o)  
    {  
        //logic  
    }  
}

- In a mapping file of hibernate this Generator class can be configured for an <id> tag like the following

```
<id name="productId" column="Pid">  
    <generator class="MyGenerator" />  
</id>
```

- The following Generator class is used for generating a product Id by using the logic a Random number from 0-10 multiply with length of the product name.

```
// MyGenerator.java  
import org.hibernate.id.*;                         → IdentifierGenerator  
import org.hibernate.engine.*;                     → SessionImplementor  
import java.util.*;                                 → Random  
import java.io.*;                                 → Serializable
```

```

public class MyGenerator implements IdentifiesGenerator
{
    public Serializable generate(SessionImplementor si, Object o)
    {
        Random r = new Random();
        int k = r.nextInt(10); // It contains 0-10 numbers line by line Serialized
        Product p = (Product)o; // it is typecast into product pojo class
        String str= p.getProductName(); // Eg: SONY → 4 letters
                                         // 1+4=5 Total length
        int length = str.length(); // SAMSUNG → 7 letters. length=7
        Integer i = k * length; // i=1*7=7
                                // =2*7=14
        return i;
    }
}

```

wrapper classes shows null value  
 but primitives show '0' value  
 so, Entity don't configure to see  
 the result. It is '0' value.  
 We have to take wrapper class  
 then it contains null value.

### Servlet with Hibernate Integration :-

- In Servlet to a DataBase communication, a Servlet can use JDBC code to talk with a Database.
- In place of JDBC code, if we use Hibernate then we call this concept as Servlet with Hibernate Integration.
- In Servlet with Hibernate Integration, we have the following 3 approaches for developing a Servlet with Hibernate persistence logic
- Approach 1:-
  - We can insert the steps for creating Hibernate client application in Service method of a Servlet.
  - If Service(-,-) method contains building SessionFactory then when the no. of

Clients are increasing then the no. of SessionFactories are also increased at the

Server Side.

SessionFactory Contains no of sessions. Each & Every request more SessionFactories will be created. one SessionFactory contains no of sessions, so, it is heavy weight obj for the one SessionFactory. If more SessionFactories will be created then the server side problem will there. It means less got

→ If no. of SessionFactories are increased then burden on a Server is also increased

Because SessionFactory of Hibernate is a Heavyweight Object.

→ This approach is not a good approach for integrating a Servlet with Hibernate.

Eg:- public class MyServlet extends GenericServlet

{

    public void service (request, response) throws SE, IOE

{

    // Configuration Object

    // configure()

    // buildSessionFactory

    // openSession

    // beginTransaction

    // commit

    // close

}

}

Approach - 2:-

→ instead of defining the SessionFactory creation in Service(-,-) method,

we can devide buildingSessionFactory into init(), sessionFactory into logic

Service() and closing the sessionFactory into destroy().

The Advantage of this approach is, SessionFactory of Hibernate is generated for

once, not for every request on a ServletObject. Bcz init() of a Servlet will be

called for once per a Servlet Object.

→ If an Application Contains multiple Servlets and if this Approach-2 is

applied then a SessionFactory is separately generated for Each Servlet

for once. It means, if an application Contains 4 servlets then

SessionFactory of hibernate is generated for four times.

→ This Approach-2 is suitable, if an Application contains Single Server to Integrate with Hibernate.

→ Eg:-

```
public class MyServer extends GenericServer  
{  
    public void init() throws SE  
    {  
        // Configuration object  
        // configure()  
        // buildSessionFactory()  
    }  
    public void service(request, response) throws SE, IOE  
    {  
        // openSession  
        // beginTransaction  
        // commit  
        // close Session  
    }  
    public void destroy()  
    {  
        // close factory()  
    }  
}
```

01/03/2013

### Approach-3:-

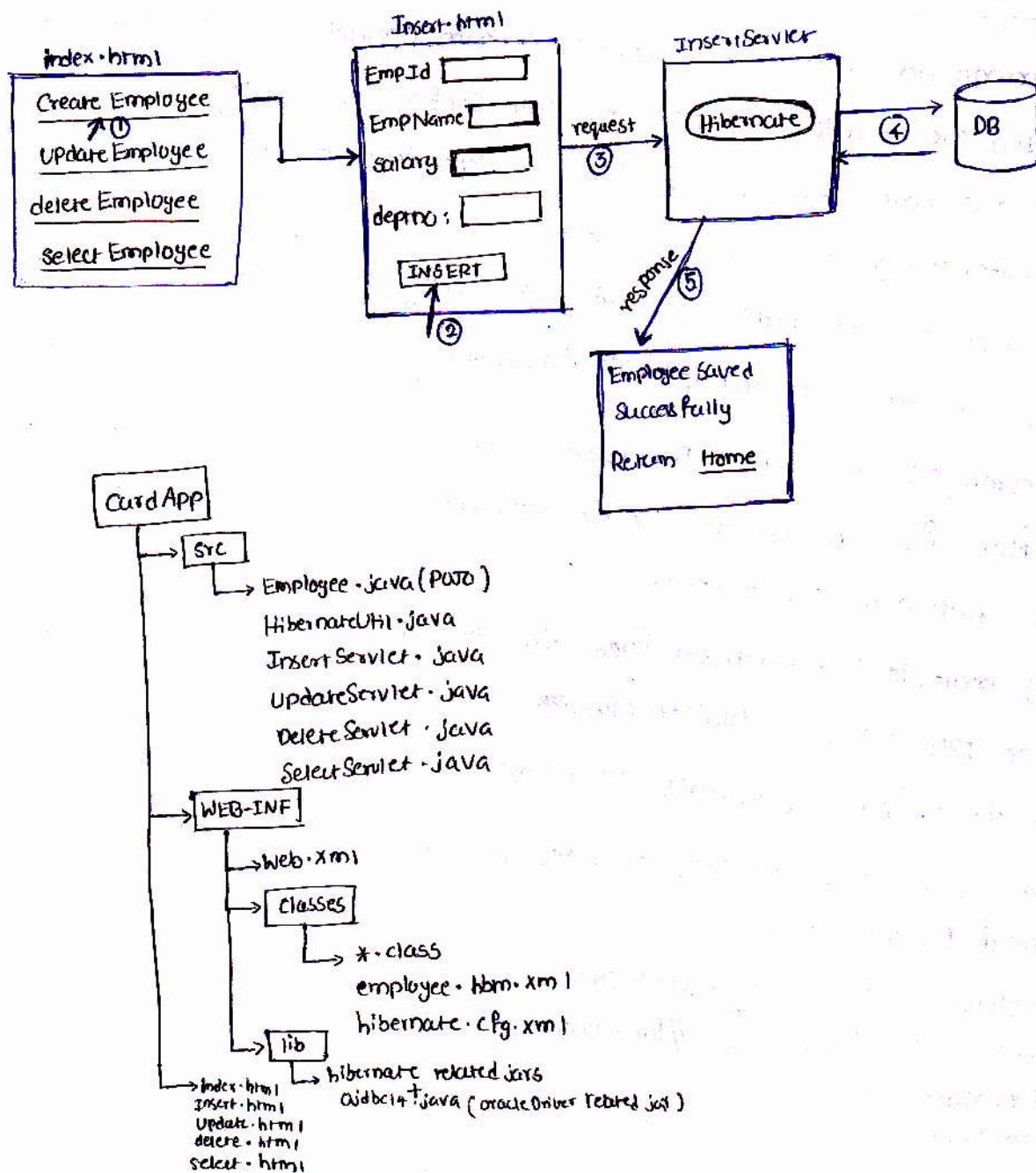
- when multiple servers are integrating with a Hibernate in a web appn then it is always better to make SessionFactory of Hibernate as Singleton and share it with all servers of the web appn.
- In order to make SessionFactory of Hibernate as Singleton, we need to create a Separate class with a StaticFactory method for building (or) generating a SessionFactory for only once.
- The following is a Utility class for making a SessionFactory of Hibernate as Singleton.

```

import org.hibernate.*;
import org.hibernate.cfg.*;
public class HibernateUtil
{
    private static SessionFactory factory;
    public static SessionFactory getSessionFactory()
    {
        if(factory==null)
        {
            factory=new Configuration().configure().buildSessionFactory();
        }
        return factory;
    }
}

```

- The following example is to perform CURD operations on Employee objects using Servlet with Hibernate Integration.



- When Integrating a Webapplication with hibernate, we need to copy POJO class, mapping file, configuration file of Hibernate into classes folder and The main and dependent jar files of Hibernate into lib folder
- To compile the .java files of the above application, we need to set the following 2 jars in the CLASSPATH.
  - (1) servlet-api.jar
  - (2) Hibernate3.jar
- Deploy the application Root Directory (CURDAPP) into <Tomcat-home>/Webapps Folder.
- Start the Tomcat Server and send the following request from a Browser.  
`http://localhost:2013/CurdApp`

02/03/2013

hibernate3.0.1/maven/etc.

### \* Connecting with Multiple databases :-

- From an Hibernate application, if we want to perform operations on Multiple Databases at a time then we need to construct multiple configuration files.
- In hibernate the no.of configuration files are depending on the no.of Databases connecting simultaneously from an application.
- In a client appn of Hibernate, to load each configuration file, we need a SessionFactory. It means for multiple configuration files we need multiple factories.
- To perform operations on multiple DataBases, we need a session with each Database and Transaction for each session. so, we need multiple sessions and multiple transactions within a client appn.
- When creating multiple configuration files for connecting with multiple databases, the connection properties and Dialect class in configuration files will be different and the remaining properties are almost same.
- For Connecting a DataBase of mysql, we need to configure the following connection properties and Dialect class.

DriverClass : com.mysql.jdbc.Driver

URL : jdbc:mysql://localhost:3306/test

Username: root1

Password: root1

dialect : org.hibernate.dialect.MySQLDialect;

→ For connecting with PostgreSQL DataBase, the following are the connection properties and the Dialect class.

Driver class : org.postgresql.Driver

UserName : postgres

password : postgres

URI : jdbc:postgresql://localhost:5432/postgres

dialect : org.hibernate.dialect.PostgreSQLDialect

NOTE:- ① If connection properties (or) Dialect class are unknown for a DataBase then we can get the information from hibernate.properties file that is distributed along with Hibernate s/w.

② The hibernate.properties file is available at  
D:\hibernate-distribution-3.6.5 Final\project\etc folder

→ Apart from Hibernate jar files, we need to set MySQL and PostgreSQL Driver related jar files in the CLASSPATH, while connecting with MySQL & PostgreSQL at a time.

→ com.mysql.jdbc-5.1.5.jar

→ postgresql-jdbc3.jar

\* To open MySQL prompt then do the following

Start → programs → MySQL → MySQL Server 5.0 → MySQL Command Line

→ Enter password : root1

mysql> connect test  
↳ DataBase name

mysql> select \* from student;

\* To open Commandline client of PostgreSQL, do the following

Start → programs → postgresql8.3 → psql to Postgres →

Enter password : postgres

Postgres=# select \* from student;

## Multiple DB Test

```
↳ Student.java  
    Student.hbm.xml  
    hibernate-mysql.cfg.xml  
    hibernate-PostgreSQL.cfg.xml  
    InsertClient.java
```

### hibernate with Connection Pooling

- While Obtain a Connection with a DataBase using a DriverManager class then DriverManager class Directly opens a connection with a DataBase Server. and it is a non reusable connection
- If a non-reusable connection is opened then after DataBase operations are completed then it must be closed. otherwise it will be still in Open state and that connection is a non-reusable for another client.
- If each time a connection is opened and then closed then this opening and closing makes burden on a DataBase server. so, the performance of the DataBase server will be decreased.
- In order to overcome the above problems, we got a technique called Connection pooling.
- In Connection pooling technique a set of connections are opened with a DataBase server and stored them in a pool. The connections in the pool are reused for multiple ~~new~~ clients.
- With the help of this reusability, the burden on a DataBase server will be reduced, so that the performance of the DataBase server will be increased.
- A Connection pool will be created and monitored by the server administrator at serverside in realtime apps.

Q:- What is the diff b/w a pool and a cache?

Ans: (a) In a pool all objects are equal objects. It means 1<sup>st</sup> object in the pool is exactly same as a last object in the pool. we also call this type of objects as stateless objects. But a cache is a group of stateful objects because in a cache no two objects are equal.

(b) In a pool, when it is busy then a client waits until one of the multiple objs in pool is free. But when a cache is busy then a client waits until a particular object is free. It means waiting time in a pool is less than waiting time in a cache.

→ By default every hibernate application uses Connection Pooling Technique internally while talking with DataBases. The built-in connection pooling in hibernate is also called JDBC connection pooling.

→ In hibernate the connection pooling is divided into the following 3 categories

- 1) Built-in JDBC connection pool
- 2) Third party connection pools
- 3) Server side connection pool.

#### ① Built-in Connection Pool:-

→ By default, hibernate creates internally a connection pool when an application execution is started.

→ By default, the built-in connection pool of hibernate contains minimum capacity as '1' and maximum capacity as '8'.

→ As a programmer we can change the maximum capacity of the Built-in Connection pool by setting (or) Configuring `hibernate.connection.pool-size` property in hibernate Configuration file.

for Eg:-

```
<property name="hibernate.connection.pool-size">15</property>
```

#### ② Third party Connection Pool:-

→ Hibernate is recommended to use the built-in JDBC Connection pool, only for testing the applications of hibernate. but it can't be used for production mode applications (Real time apps).

→ While creating Realtime applications with hibernate and without a server environment then hibernate is recommended to use any one of the following two third party OpenSource Connection pools, which are distributed along with hibernate s/w.

① C3P0 connection pool

② proxool connection pool

① C3P0 connection pool :-

→ C3P0 is an open source third party connection pool, it is distributed along with hibernate in the form of a jar file C3P0-version.jar.

→ To activate this C3P0 connection pool, we need the following two changes.

① add C3P0 related properties to the configuration file of hibernate.

(hibernate.cfg.xml)

② add C3P0-version.jar in the CLASSPATH.

→ For C3P0 connection pooling, we need to add the following 3 properties in the Configuration file.

<property name="hibernate.connection.provider\_class"> org.hibernate.connection.  
C3P0ConnectionProvider </property>

<property name="hibernate.c3p0.min\_size"> 2 </property>

<property name="hibernate.c3p0.max\_size"> 10 </property>

→ C3P0 jar file is available at the following location.

<hibernate-home>\lib\optional\C3P0\c3p0-version.jar.

② proxool connection pool :-

→ This proxool connection pool is also an open source third party connection pool, it is distributed along with hibernate in the form of a jar file proxool-version.jar

→ When this proxool connection pool is activated then automatically the builtin JDBC connection pool will be deactivated.

→ To activate this proxool Connection pool, the following are the 3 changes

required.

① Create a separate XML file with the connection properties and maxsize of the pool. This XML filename can be <anyname>.xml

② Configure the separate XML file created and along with it the provider class and alias name, in hibernate configuration file.

③ While executing, set proxool-version.jar in the CLASSPATH

→ While creating a separate XML file for a proxool it doesn't require any DTO for the XML and the root element of XML can be any name.

→ In proxool connection pool minimum size of the pool is '1' and we can't set minsize explicitly. But we can set maxsize of the pool.

<!-- proxool.xml -->

<something>

<proxool>

<alias> abcd </alias>

<driver-url> jdbc:oracle:thin:@localhost:1521:orcl </driver-url>

<driver-class> oracle.jdbc.OracleDriver </driver-class>

<driver-properties>

<property name="user" value="scott"/>

<property name="password" value="tiger"/>

</driver-properties>

<max-connections-count> 5 </max-connection-count>

</proxool>

<something>

→ In hibernate configuration file we need to remove connection properties and in place of them we need to add proxool properties.

<!-- proxool properties -->

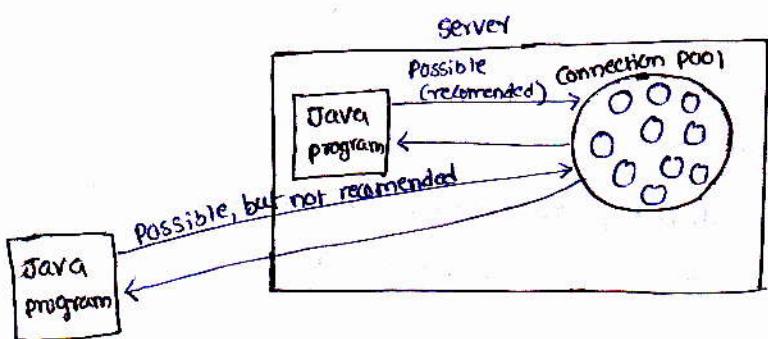
<property name="hibernate.proxool.xml"> proxool.xml </property>

<property name="hibernate.proxool.pool\_alias"> abcd </property>

<property name="hibernate.connection.provider-class"> org.hibernate.Connection.ProxoolConnectionProvider </property>

### 3) Server-side connection pool :-

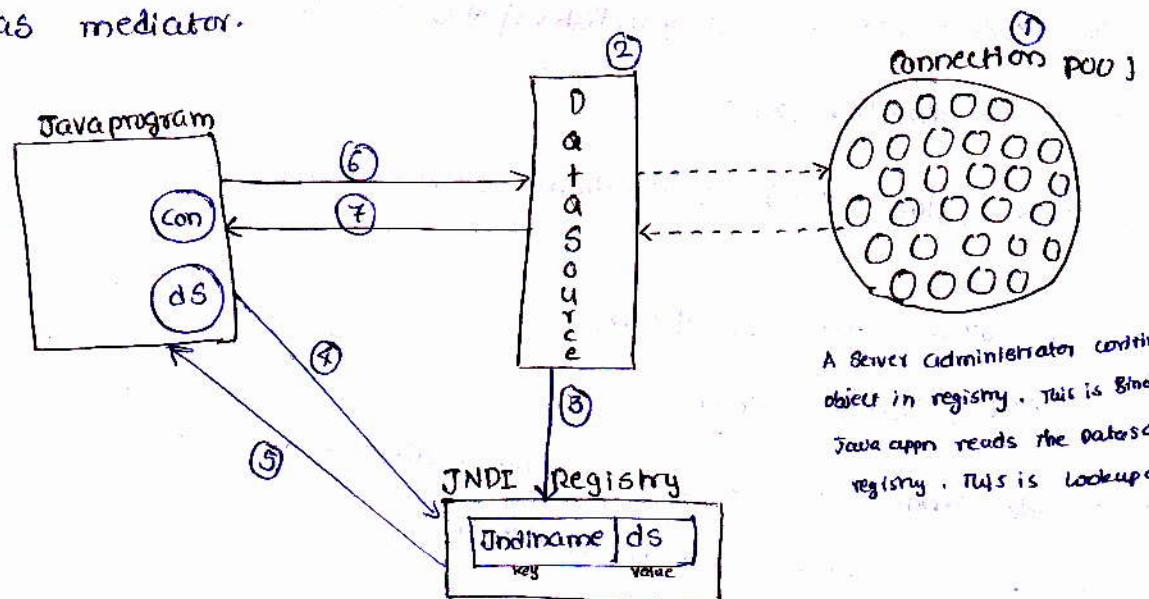
- In hibernate, when an application is running at serverside, it is recommended to use a ServerSide Connection pool configured by the Server administrator.
- Actually if a connection pool is configured at server-side then a Java application running either with in the Server or an application running at out of the server can access that connection pool.
- If a Java application running at out of the server then to access a connection from pool, the server must be started. If a Java application is running inside the server then it is recommended to use a Server Side connection pool.



→

- At serverside, an Administrator configures a connection pool and also a DataSource.
- A DataSource acts like a mediator between a Java program and a Connection pool.
- A Java program will ask a DataSource to obtain a Connection from a Pool. (get)
- In the middle a ~~Java program~~ DataSource ~~get~~(obtains) the connection from pool, then creates proxy connection and then that proxy connection will be send to Java program.
- A proxy connection is also called as a logical connection and it is not a real connection, but acting like a real connection in our Java program.
- When a Java program closes the Java program then internally the real connection (or) (pool connection) sent back to the pool.

- A Server administrator stores DataSource object in Jndi registry of the server. A Jndi registry is a common area, to share the objects ~~with~~ between programmers and administrators.
- A Java program first connects with registry and takes DataSource object and then obtains a connection from a pool through DataSource object as mediator.



#### Writto Diagram

- ① A Server administrator Configure a Connectionpool in a server, for a DataBase.
  - ② A server administrator Configure DataSource as a mediator for a Connection pool.
  - ③ An administrator Registered (stored) DataSource object in JNDI registry.  
This is also called "Binding".
  - ④ & ⑤ A Java program uses Jndi properties and connects with registry and obtains DataSource object, by using its Indiname (key). This is also called "Lookup operation".
  - ⑥ & ⑦ A Java program asks for a connection by calling getConnection() method on DataSource object, and finally obtains a connection from the pool.
- In case of hibernate, hibernate internally obtains a DataSource from registry and then obtains a connection from the pool through DataSource object. For this, we need to remove Connection properties and instead of them we need to attach Jndi properties in hibernate.cfg.xml.

- ① hibernate-connection.datasource → JndiName (key)
- ② hibernate.jndi.class } depends upon the server like Glassfish Server (or) Weblogic Server
- ③ hibernate.jndi.url

05/08/2013

### Configuring a Connection pool in Glassfish V3 Server :-

Step-I:- copy ojdbc14.jar in C:\glassfishv3\glassfish\domains\domain1\lib\ext folder.

Step-II:- start the glassfish server

```
C:\glassfishv3\glassfish\bin> asadmin start-domain domain1
```

Step-III:- open the browser and type the url

http://localhost:4848

Step-IV:- At left side expand Resources → expand JDBC → select Connection Pools →

click **New** button → enter the following details

Name: **poolone**

Resource Type: **Javax·sql·DataSource**

Database vendor: **Oracle**

→ click **Next** button

Step-V:- Enter the following 3 additional properties

Passsword	tiger
URL	JDBC:oracle:thin:@localhost:1521:orcl
User	Scott

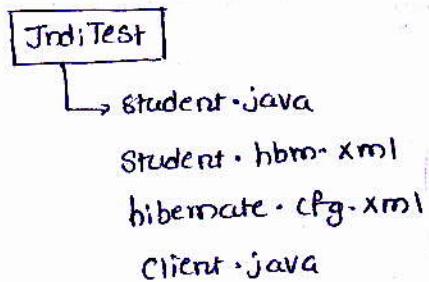
click on **Finish** button

Step-VI:- At left side select JDBC Resources → New → Enter to **JNDIName: Jndi1**

→ Pool Name **poolone** → OK

NOTE:- To Verify whether a Connection pool is successfully created or not, click on **poolname** → **Ping**

The following example is to obtain a connection from a Connection pool of GlassFish server for storing a Student object in the Database using Hibernate.



For this example we need to set ~~use~~ the following 3 jar files of GlassFish server, apart from all the jar files of hibernate in the CLASSPATH.

- ① appserv-rt.jar (c:\glassfishv3\glassfish\lib)
- ② javaee.jar (c:\glassfishv3\glassfish\lib)
- ③ imqjmsra.jar (c:\glassfishv3\glassfish\lib\install\application\jmsra)

for this example refer page no: ⑧ an application ④ of the Handout.

NOTE:- When executing the above example, if we get an exception with message ~~SIP4J API~~, SIP4J-API-1.6.0 (or later) is incompatible then set SIP4J-API-JAR in classpath by removing SIP4J-API-1.6.0.jar from the CLASSPATH. (we can download SIP4J-API-JAR from hibernate-3.4)

### Configuring a Connection pool using weblogic 10.x:

Step-I:- Create a server domain (if no domain exist)

Start → programs → Oracle WebLogic → weblogic server → TOOLS → Configuration Wizard

Step-II:- ① create a new WebLogic domain → Next → Next → Enter

Domain name:

User name:

Password:

Conform password:

→ next → next → next → create → Done

Step-III:- Start the Weblogic Server

C:\Oracle\Middleware\user\_projects\domains\hibernate\_domain\startWeblogic.bat

Step-IV:- open the Browser and Type the following URL in address bar

http://localhost:7001/console

Username : Weblogicadmin

Password : Weblogic10

Login

Step-V:- Expand Services at leftside → expand JDBC → select Datasources →

New → Enter the following  
poolname

Name : poolone

JNDI Name : oraclejndi

Database Type : Oracle

Database Drivers : Oracle's Driver (Thin) For Service Connections

→ next → next → Enter

the following connection properties

Database Name : Oracle

HostName : localhost

port : 1521

Database Username : Scott

Pass word : Tiger

Confirm password : Tiger

Next

Step-VI:- click on Test Configuration → Next →  AdminServer → Finish

NOTE:- if you want to set capacity for a connection pool then click on poolone (poolname)

→ select connectpool tab → Enter the capacity InitialCapacity : 1 → save  
maximumCapacity : 10

In the previous example, if we want to use Connection pool of WebLogic Server then in hibernate.cfg.xml file, we need to add the following Jndi properties

```
<property name="hibernate-connection.datasource"> oraclejndi </property>
```

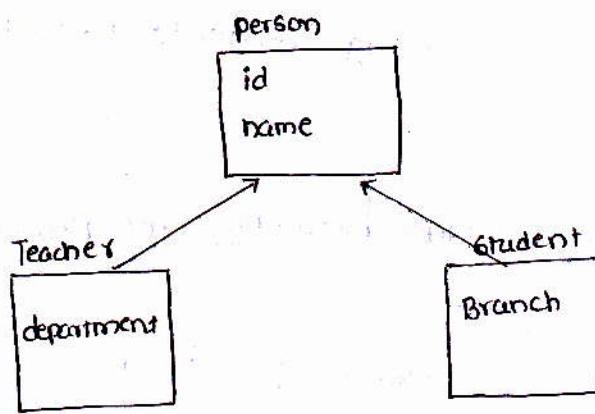
```
<property name="hibernate.jndi.class"> WeblogicJndi.WLInitialContextFactory </property>
```

```
<property name="hibernate.jndi.url"> t3://localhost:7001 </property>
```

In CLASSPATH we add the following one additional jar file called WebLogic.jar (C:\oracle\middleware\wlserver-10.3\server\lib).

## Inheritance mapping in hibernate

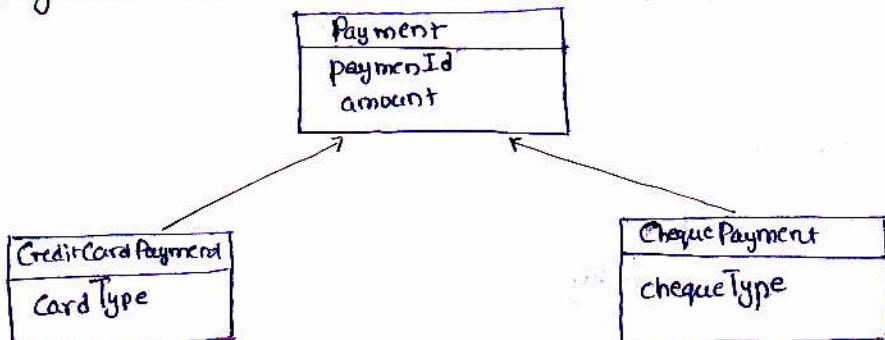
- While creating applications in Java, if more than one class is going to have some common attributes then instead of repeating the common attributes individually in each class, we apply inheritance mechanism by separating generalized attributes and specialized attributes.
- For example, we have two classes called Teacher and Student with common attributes as id and name. and special attributes as department in Teacher and branch in Student. Then we can convert this application into inheritance by creating a parent class (or) Base class has a person like the following

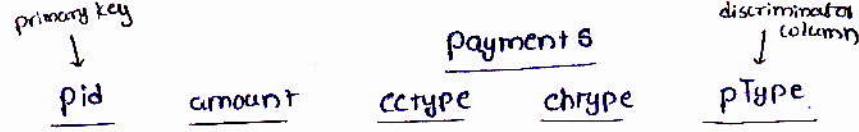


- While persisting the inherited objects i.e. a Teacher (or) a Student object then hibernate persists the inherited properties also in the DataBase. It means both generalized and specialized attributes in the DataBase. So, we say that hibernate supports inheritance.
- To inform the Hibernate about the generalized and specialized properties of an object, we need to add some additional configuration in a mapping file (hbm file). This is called inheritance mapping in hibernate.
- Inheritance mapping in hibernate is also called Hibernate hierarchies.
- Hibernate has provided the following 3 strategies to represent the inherited data in DataBase.
  - ① Table per class hierarchy
  - ② Table per subclass hierarchy.
  - ③ Table per Concrete class hierarchy.

## ① Table per class hierarchy :-

- In this type of strategy, hibernate uses a single Table in DataBase for representing the entire Objects of a class hierarchy of a Java application.
  - In other words, if Table per class hierarchy, the Objects of all derived classes are stored in a single table DataBase.
  - In this Type of hierarchy, a DataBase table contains columns for generalized attributes, columns for specialized attributes and an additional column called Discriminator column.
  - A Discriminator column stores a Discriminator value, which is used to identify one Derived class object among multiple Derived classes easily in a DataBase.
  - While working with this Table per class hierarchy, the Discriminator column is mandatory in a DataBase table.
  - Table per class hierarchy can't be implemented by without using Discriminator column.
  - While creating a Single table, the columns related to Specialized attributes (Derived classes attributes) should not be applied with not null constraint.
  - A Discriminator column stores a Discriminator value and it is used to identify one Derived class object among multiple Derived classes objects stored in a single table.
- Eg:- ① We have a Base class called payment, with two derived classes creditCard payment and cheque payment.
- ② we are applying Table per class hierarchy for storing both credit card and cheque payment objects in a Single table of DataBase.





→ In hibernate mapping file, we need to use `<subclass>` tag for mapping subclasses of the hierarchy.

→ We have two approaches for mapping the subclasses.

① We can configure the Base class and the subclasses in a single Mapping file.

② We can separate Base class and each subclass configuration into a different mapping file.

In the above two approaches, the 1<sup>st</sup> approach is preferable, Bcz it reduces the no. of XML files (mapping files). Apart from this, in a Configuration file we no need to configure mapping tag for multiple times.

#### 1<sup>st</sup> approach:-

`<!-- Payment.hbm.xml -->`

`<hibernate-mapping>`

`<class name="Payment" table="payments">`

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

`*** <discriminator column="ptype" type="java.lang.String" />`

`<property name="amount" />`

`*** <subclass name="CreditCardPayment" discriminator-value="credit" />`

`<property name="cardType" column="cctype" />`

`</subclass>`

`<subclass name="ChequePayment" discriminator-value="cheque" />`

`<property name="chequeType" column="chrype" />`

`</subclass>`

`</class>`

`</h-m>`

#### II<sup>nd</sup> approach:-

`Payment.hbm.xml`

`<h-m>`

`<class name="Payment" table="payments" >`

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

```

<discriminator column="Pmode" type="java.lang.String"/>
<property name="amount"/>
</class>
</h-m>

Credit.hbm.xml
<h-m>
<subclass name="CreditCardPayment" extends="Payment" discriminator-value="credit">
<property name="cardType" column="cctype"/>
</subclass>
</h-m>

Cheque.hbm.xml
<h-m>
<subclass name="ChequePayment" extends="Payment" discriminator-value="cheque">
<property name="chequeType" column="chtype"/>
</subclass>
</h-m>



|                        |
|------------------------|
| <b>TablePerClass</b>   |
| → Payment.java         |
| CreditCardPayment.java |
| ChequePayment.java     |
| Payment.hbm.xml        |
| hibernate.cfg.xml      |
| InsertClient.java      |
| *.class                |


```

After executing this table per class client application then the following outputs generated in DataBase.

SQL> Select \* from payments;

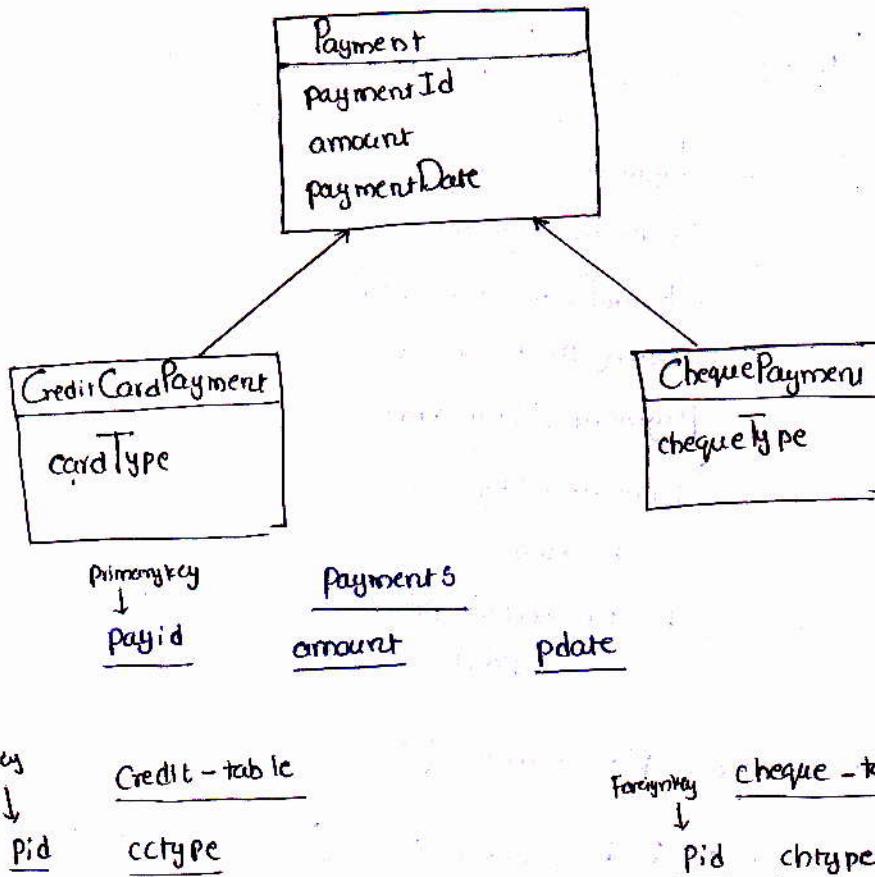
PAYID	Amount	PDATE	CCTYPE	CHTYPE	PMODE
1	5000	07-MAR-13	VISA		CR
2	8000	07-MAR-13		ORDER	CT

## ② Table Per SubClass Hierarchy :-

- In this type of hierarchy, hibernate represents the inheritance data in separate tables, but not in a Singletable.
- In this type of inheritance mapping, hibernate uses a separate table for storing Base class attributes and a separate table for each derived class for storing the Derived class attributes.
- In order to maintain the relationship between the data stored in Base class table and Derived class tables, hibernate uses Foreignkey relationship.
- In this Tableper subclass hierarchy, the use of discriminated column is optional.
- In a hibernate mapping file, we need to use a tag <joined-subclass> for configuring Table per subclass hierarchy.

### Example :-

We have a Baseclass called Payment with two derived classes CreditCardPayment and ChequePayment. we are applying TablePerSubclass hierarchy for representing this data in a DataBase by hibernate.



## Payment . hbm . xml

<hibernate-mapping>

<class name="Payment" table="payments">

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

<property name="amount"/>

<property name="paymentDate" column="pdate"/>

another table in  
this inheritance tree  
this reg  
\*\*\*\*<joined-subclass name="CreditCardPayment" table="credit\_table">

\*\*\*\*<key column="pid"/> → inform to Hibernate this indicates foreign key column

<property name="cardType" column="ctype"/>

<joined-subclass>

\*\*\*\*<joined-subclass name="ChequePayment" table="cheque\_table">

\*\*\*\*<key column="p\_id"/>

<property name="chequeType" column="ctype"/>

<joined-subclass>

<class>

</hibernate-mapping>

Hibernate will automatically copies the primary key value into Foreign key.

### TablePerSubclass

→ Payment . java

CreditCardPayment . java

ChequePayment . java

InsertClient . java

Payment2 . hbm . xml

hibernate . cfg . xml

For this example refer  
page no : 10 appn ⑥  
of the Handout.

Create the following tables at Database :-

\* . class

SQL > create table payments ( payid number(5) primary key, amount number(9,2),  
pdate date );

SQL > create table credit\_table ( pid number(5), ctype varchar2(10) );

SQL > create table cheque\_table ( p\_id number(5), ctype varchar2(10) );

SQL > select \* from payments ;  
SQL > select \* from credit\_table ;  
SQL > select \* from cheque\_table ;

```
SEL> select * from payments;
```

payID	Amount	pdate
1	5000	08-may-2013
2	8000	08-may-2013

```
select * from credit-table;
```

PIP	CCTYPE
1	VISA

```
select * from cheque-table
```

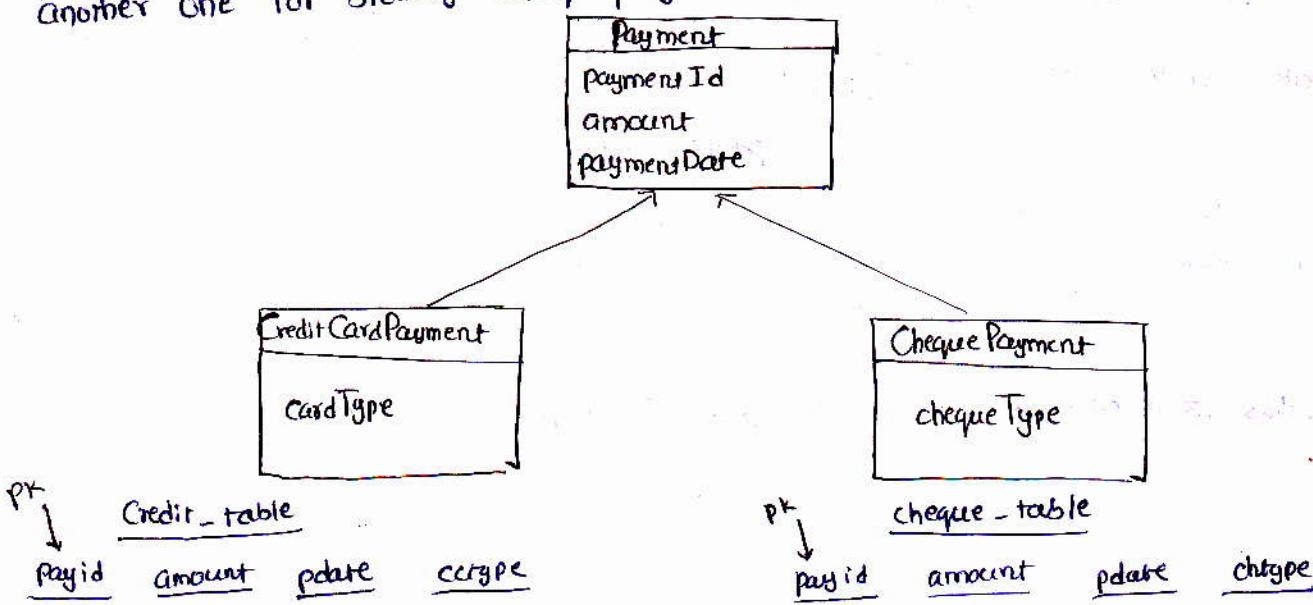
PIP	CHTYPE
2	ORDER

### ③ Table Per Concrete class hierarchy :-

- In this type of Inheritance hierarchy, hibernate uses a separate table for storing the generalized and specialized attributes of a derived class.
- In this type of Inheritance mapping, hibernate uses a separate table in DataBase for each Derived class of the hierarchy.
- In this Type of Inheritance mapping also, discriminator column is optional.
- In a mapping file of Hibernate, we need to configure `<union-subclass>` tag for Configuring TablePerConcrete class hierarchy.

```
<subclass> -----> Tableperclass  
<joined-subclass> -----> Table per subclass  
<union - subclass> -----> Table per Concrete class
```

Example:- we have a Base class called payment with two derived classes CreditCardPayment and ChequePayment. In TablePerConcrete class hierarchy, hibernate uses two tables in DataBase, one for storing CreditCardPayment objects another one for storing ChequePayment objects.



### payment3.hbm.xml

```

<h-m>
  <class name="Payment">
    <id name="paymentId" column="pay_id"/>
    <property name="amount"/>
    <property name="paymentDate" column="pdate"/>
    *** <union-subclass name="CreditCardPayment" table="credit-table">
      <property name="cardType" column="ccType"/>
    </union-subclass>
    *** <union-subclass name="ChequePayment" table="cheque-table">
      <property name="chequeType" column="chtType"/>
    </union-subclass>
  </class>
</h-m>

```

#### TablePerConcreteClass

→ payment.java

CreditCardPayment.java

ChequePayment.java

InsertClient.java

payment3.hbm.xml

hibernate.cfg.xml

\*-class

For this example refer  
Page: (1) config (7).

SQL> create table credit-table (payid number(5) primary key, amount number(9,2),  
pdate date, ccType varchar2(10));

SQL> create table cheque-table (payid number(5) primary key, amount number(9,2),  
pdate date, chType varchar2(10));

SQL> Select \* From credit-table;

PAYID	AMOUNT	PDATE	CCTYPE
1	5000	08-MAR-13	VISA

SQL> select \* from cheque-table;

PAYID	AMOUNT	PDATE	CHTYPE
2	5000	08-MAR-13	ORDER

In this 3 inheritance better to choose Table per Concrete class

## HQL (Hibernate Query Language)

09/03/2013

### Bulk Operations in Hibernate :-

- While working with Hibernate, operations on objects <sup>in hibernate</sup> can be done in two ways.
- that is 1) Single row operations and 2) Bulk Operations.
- While performing Single row operation, the operation will be effected on a single row of the Database.
- In case of BulkOperations, the operations are effected on multiple rows at a time on DataBase
- Hibernate Framework has provided the following 3 techniques for performing Bulk Operations.
- ① HibernateQuery Language
- ② Criteria API
- ③ Native SQL

### HQL:-

- An Object oriented form of SQL is called HQL.
- Object Oriented form is nothing but a query which is constructed by replacing Table name with pojo class name and column names with pojo class Variable names.
- The big difference b/w writing SQL and HQL queries is, SQL queries are DataBase dependent, but HQL queries are DataBase Independent.
- We can perform both Select and non-select operations on a DataBase using HQL.

## Select Operation in HQL:-

→ Constructing a select operation in HQL is almost equal to constructing

### Select operation in SQL.

→ public class Employee

{

→ employeeId

→ employeeName

→ employeeSal

→ deptNumber

employee e

empno

ename

sal

deptno

### Eg(1):-

sql → select \* from employee

hql → select e.\* from Employee e

(or)

from Employee e

→ While using above HQL query, we can select all employees from the Database at a time. In hibernate terminology, if we load ~~only~~ complete details of a row then it is called loading Full object from the Database.

→ While loading Full object, we can directly start an HQL query with "from" keyword.

### Eg(2):-

sql → select ename, sal from employee

hql → select ~~ename~~, e.employeeName, e.employeeSal from Employee e

→ by using the above HQL query, while loading the partial details of employee from the Database. It is called loading partial objects.

→ While loading partial objects, we can't start an HQL query with "from" keyword.

### Eg(3):-

sql → select \* from employee where deptno = 20

hql → select e.\* from Employee e where e.deptNumber = 20

from Employee e where e.deptNumber = 20

- By using the above HQL query, the employees working in 20<sup>th</sup> department are selected. In the above query, Full Objects are loaded from the DataBase, so we started HQL query with "from" keyword.

Eg(4):-

sql → select ename, sal from employee where deptno=?

hql → Select e.employeeName, e.employeeSal from Employee e Where e.deptNumber=?

Select e.employeeName, e.employeeSal <sup>from Employee e</sup> Where e.deptNumber=:p1

- by using the above HQL query, we are loading partial employee objects,

by depending on given deptNumber given at runtime

- in an HQL query, to put a runtime value, we can insert either a positional parameter (or) a Named parameter into the query.

positional parameter (?)

Named parameter (:<name>)

- For executing HQL queries, we need a Query reference

to get a Query reference, we need to call createQuery() of Session Interface

- Session Interface in Hibernate is the main runtime interface b/w a programmer and Hibernate. Because all the methods required for a programmer to talk with hibernate are given by Session Interface of hibernate.

Syntax :-

Query qry = session.createQuery("hql command");

- After constructing a Query reference, we need to call one of the following two methods for executing that query.

① list() → for select operation

② executeUpdate() → for non-select operation

→ Query is an interface of org.hibernate pkg and its reference points to its Implementation class object.

→ The implementation class of Query Interface is **QueryImpl**

→ There are 3 ways of executing a select operation of HQL.

1<sup>st</sup> way :-

→ In this way we are selecting Full objects from DataBase

→ While selecting Full objects from DataBase by using HQL query, internally hibernate performs the following.

① Hibernate reads the rows from a table and stores them in a ResultSet object.

② Hibernate reads each row of ResultSet and stores the data in a POJO class object

③ Hibernate stores all pojo objects in a Collection of Type ArrayList (java.util.List)

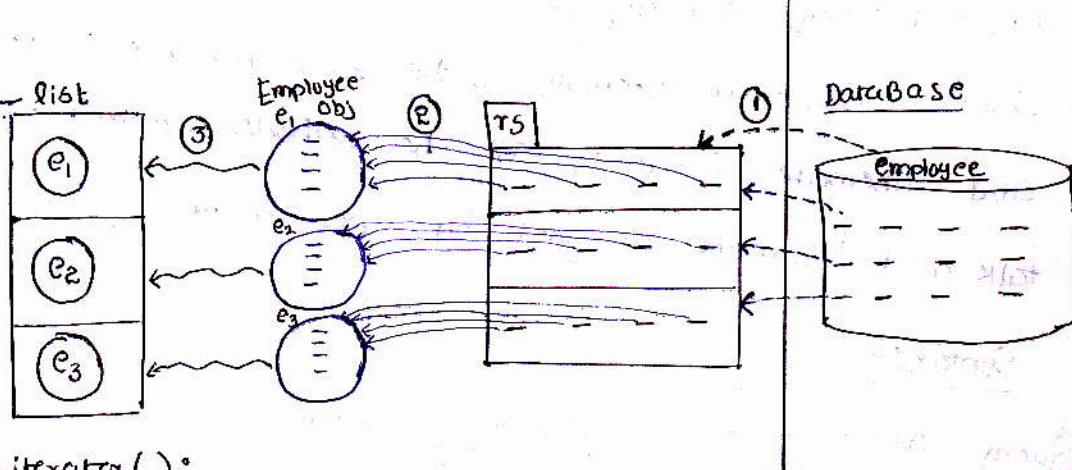
④ Finally hibernate returns the Collection reference back to our Java programs.

→ After obtaining a reference of List collection then we need to iterate that collection for reading one by one object from the collection.

example :-

```
Query qry = session.createQuery ("from Employee e");
```

```
List list = qry.list();
```



```
Iterator it = list.iterator();
```

```
while (it.hasNext ())
```

```
{
```

```
Object o = it.next();
```

```
Employee e = (Employee) o;
```

```
s.o.p (e.getXxx());
```

```
}
```

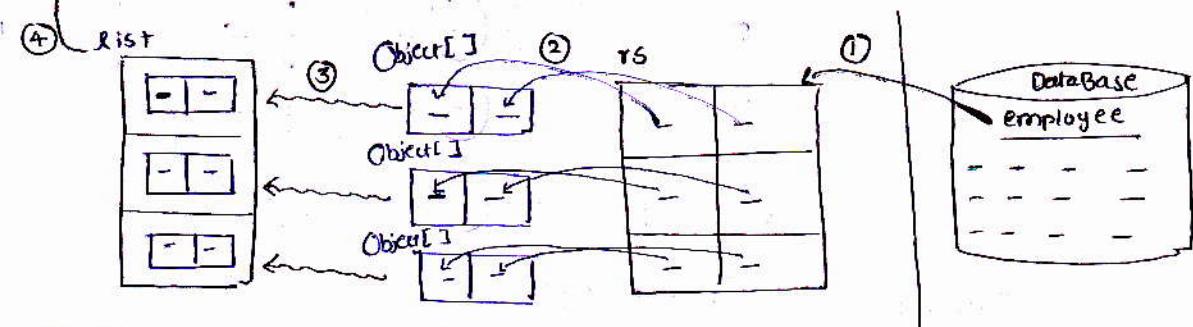
### Way-2:-

- In this way, we are reading (or) loading partial objects from the DataBase using HQL query.
- While reading partial objects, Hibernate internally performs the following
  - Hibernate reads the Data from DataBase table and stores it in a ResultSet object
  - Hibernate stores each row data of ResultSet into an Object[]
  - Hibernate stores Object[] into java.util.List collection
  - Finally Hibernate returns the list collection back to our Java program.
- While iterating the collection, we need to typecast each element of the collection into Object[]

### Example:-

- Query qry = session.createQuery ("select e.employeeName, e.employeeSal from Employee e")

```
List list = qry.list();
```



```
Iterator it = list.iterator();
```

```
while (it.hasNext())
```

```
{
```

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

```
S.O.P (row[0] + " " + row[1]);
```

```
}
```

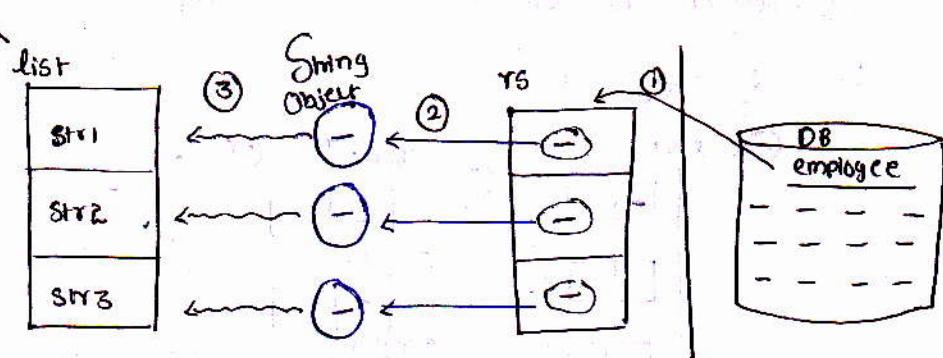
### Way 3:-

- In this way, we are reading a single property of an Object (or) objects from the DB using HQL query.
- While executing this type of query then internally hibernate performs the following

- ① Hibernate reads the data from table and stores in ResultSet.
  - ② Hibernate finds the property type using ResultSet metadata and stores the value of each row of ResultSet into that Particular type of object.
  - ③ Hibernate stores the objects into a collection of type java.util.List
  - ④ Finally, Hibernate returns the collection back to our Java program
- While iterating the collection, we typecast each element of the collection into that particular Object type.

Query qry = session.createQuery("select e.employeeName from Employee e");

List list = qry.list();



Iterator it=list.iterator();

while(it.hasNext())

{

String s1=(String)it.next();

S.o.p(s1);

}

**HqlSelect**

→ Employee.java

employee.hbm.xml

hibernate.cfg.xml

SelectClient.java

\*.class

refer page no (11) appn no: (8) of handout.

## Non-select operations using HQL :-

- Using HQL, we can perform both select and non-select operations on a DataBase
- The way of constructing a non-select operation is again similar to an SQL non-select operation. But we replace a Tablename with a ~~pojo class name~~ and columns names with POJO class variables names.
- For executing a non-select operation of HQL, we need to call executeUpdate()
- While executing a non-select operation, we need to put a Transaction because hibernate needs a Transaction for non-select operations.
- For example, if we want to perform an Update operation for Updating the salary of employees, the following code is required

```
Query qry = session.createQuery ("update Employee e set e.employeeSal=800
where e.deptNumber = 20");
```

```
Transaction tx = session.beginTransaction();
```

```
int k = qry.executeUpdate();
```

```
tx.commit();
```

```
s.o.p (k + " rows updated");
```

## HQL Insert :-

- HQL Insert operation is not a straight forward operation. It means by using HQL, we cannot perform direct insert operation into a DataBase Table.
- In HQL Insert operation is nothing but copying one object data into another object.
- While constructing an Insert operation in HQL, we use insert & select queries combinedly.
- While constructing an Insert operation of HQL, the attributes of destination object and the attributes of source object, must be ~~specified~~ individually specified in the query.

For example:-  
no. of variables are equal to the source class and destination class otherwise error will come.

Query qry = session.createQuery (" insert into TestEmployee (empId, empName, empSal, deptno)  
Select e.employeeId, e.employeeName, e.employeeSal ,  
e.deptNumber from Employee e");

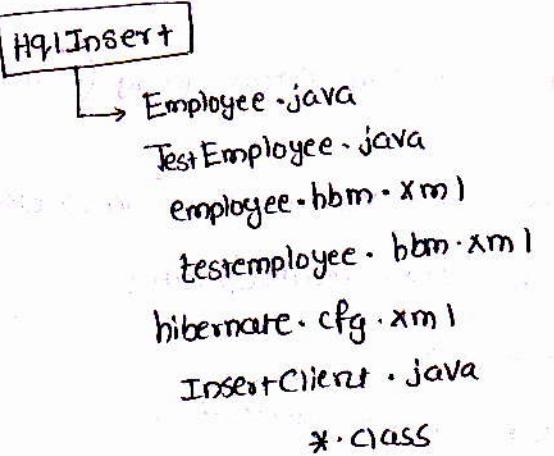
Transaction tx = session.beginTransaction();

int k = qry.executeUpdate();  
tx.commit();

→ While using the above query objects of Employee pojo class are copied to TestEmployeePojo class.

→ In HQL insert operation is nothing but copying the rows of one table into another table of DataBase.

refer page no (12) appn (9) of  
the Handout.



SQL > Select count(\*), sum(sal), max(sal) From employee

HQL > select count(\*), sum(e.employeeSal), max(e.employeeSal) from Employee e;

## Criteria API

(selecting)

- Criteria API in Hibernate is used for loading entities (objects) from the DataBase
- Using this Criteria API, only Select operation can be done. It can't perform any insert, update (or) delete operations on DataBase.

- In Criteria API, hibernate internally uses reflection mechanism for reading metadata of a pojo class. So, we need to pass Class object of a pojo class for Hibernate.

- While creating Criteria reference, we don't pass any query as a parameter. Instead of query, we pass Class object.

- To obtain a Criteria reference, we need to call createCriteria() of Session interface

Session interface  
Class clazz = Employee.class;

- Criteria crit = session.createCriteria(Class clazz);
- Criteria is preferred in hibernate, when entities (objects) are need to be loaded with multiple condition (Criteria).

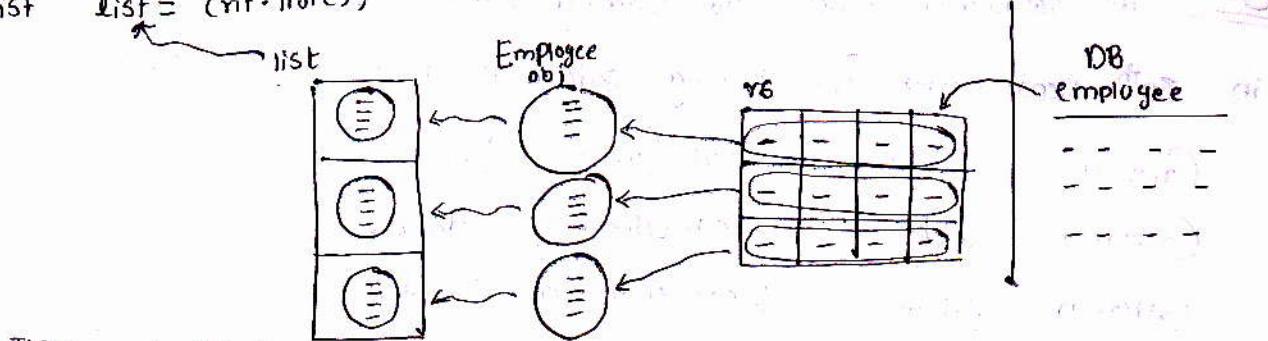
- After getting a reference of Criteria, we need to call list() method, for executing the select operation using Criteria.

List list = crit.list();

- The following is code using Criteria, for loading Full objects (or) Full object from the DataBase.

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

List list = crit.list();



```
Iterator it=list.iterator();
while (it.hasNext())
{
    Employee e=(Employee) it.next();
    S.O.P(e.getXXX());
}
```

### Adding Condition to Criteria :-

- While loading entities (objects) from a DataBase using criteria, if we want to apply a condition for loading objects then we need to create a Criterion object and then we need to add Criterion object to Criteria Object.
- Criterion is an interface and we can get its reference through Restrictions class.
- Restrictions class is not a Derived class of Criterion interface. Restrictions class is a POJO class.
- The Restrictions class contains all static methods and each static method returns a Criterion Object.
- The only similarity b/w Criterion and Restrictions, both are from Same package called org.hibernate.Criterion.

#### Eg:-① :-

The following code is for loading entities of Employee POJO class, whose name contains a letter "S", using Criteria.

```
Criteria crit = session.createCriteria(Employee.class);
Criterion condition = Restrictions.ilike("employeeName", "%s%");
```

```
Crit.add(condition);
```

```
List list = crit.list();
```

```
Iterator it = list.iterator();
```

#### Eg:-② :- The following code is for loading entities of Employee POJO class, working in 30<sup>th</sup> Department and having salary is more than 5000, using Criteria.

```
Criteria crit = session.createCriteria(Employee.class);
Criterion condition1 = Restrictions.eq("DeptNo", 30);
Criterion condition2 = Restrictions.gt("employeeSal", 5000);
Criterion condition3 = Restrictions.and(condition1, condition2);
```

```
Crit.add(condition3);
```

```
List list = crit.list();
```

To put more than one condition, first we need to make the two conditions as single condition by calling and(<sup>&</sup>) or(<sup>||</sup>) methods and then the final condition we need to add for Criteria.

Eg(3):- The following code is for loading entities of Employee POJO class, by applying the conditions as department number 30, salary should be greater than 5000 and employee name should contain 'S' letter

```
Criteria crit = session.createCriteria(Employee.class);
Criterion condition1 = Restrictions.eq("deptNumber", 30);
Criterion condition2 = Restrictions.gt("employeeSal", 5000);
Criterion condition3 = Restrictions.like("employeeName", "%S%");

Criterion condition4 = Restrictions.and(condition1, condition2);
Criterion condition5 = Restrictions.and(condition4, condition3);

crit.add(condition5);
List list = crit.list();
```

#### Adding Sorting Order to Criteria :-

→ While loading entities (Objects) from a DataBase using Criteria, it is possible to apply some sorting order for loading the objects.

→ If we want to apply a sorting order then we need to add Order object to the criteria by calling addOrder() method.

→ We can obtain an Object of Order class by calling static method of Order class only. Order is a class having two static methods asc(-) and desc(-) methods. This Order class is from org.hibernate.Criterion package.

→ The following code is for loading entities of Employee POJO class, in Descending Order of their salaries.

```
Criteria crit = session.createCriteria(Employee.class);
Order order = Order.desc("employeeSal");
crit.addOrder(order);
List list = crit.list();
-----
```

### Loading Partial Objects using Criteria:-

- While loading objects from a DataBase using Criteria, it is possible for loading particular properties of objects, instead of Complete objects from a DataBase. This is called loading partial objects.
- For loading partial objects, we need to create a Projection object for each property.
- We can obtain a projection object by calling Static methods of Projections class.
- Projections class is a factory for producing Projection objects.
- NOTE:- Similarly Restrictions class is a factory for producing Criterion objects.
- Loading partial objects using Criteria involves the following 3 steps
  - ① Create a Projection Object for each property.
  - ② Add Projection Objects to ProjectionList
  - ③ setProjectionList to Criteria.
- Finally we need to call list() for executing the Criteria.
- At the time of Iterating that Collection, we need to typecast the result into Object[].
- The following example code is for loading two properties of Employee objects from the DataBase using Criteria.

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

// Step-I

Projection p<sub>1</sub> = Projections.property("employeeName");

Projection p<sub>2</sub> = Projections.property("employeeSal");

// Step-II

ProjectionList plist = Projections.projectionList();

plist.add(p<sub>1</sub>);

plist.add(p<sub>2</sub>);

// Step-III

crit.setProjection(plist);

List list = crit.list();

Iterator it = list.iterator();

while (it.hasNext())

{

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

S.O.P(row[0] + " " + row[1]);

}

\* Loading a Single property of an Object using Criteria :-

- While loading a Single property, we no need of using ProjectionList.
- In case of loading a Single property, we can directly add a projectionObject to Criteria.
- The following code is for loading a Single property called employeeName from DataBase using Criteria.

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

Projection p = Projections.property("employeeName");

crit.setProjection(p);

List list = crit.list();

Iterator it = list.iterator();

while (it.hasNext())

{

```
String str = (String) it.next();
System.out.println(str);
```

- In hibernate, Projections concept is also used, when we want to find any aggregate functions result using criteria.
  - Finding aggregate results is nothing but like sum of salaries , count of objects (or) max of salaries etc...

The following sample code is for finding rowCount, sum & maximum of salaries of employees using Projections of Criteria.

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

111

Projection P<sub>1</sub> = Projections. rowCount();

Projection  $P_1 = \text{Projections} \cdot \text{sum}(\text{"employeeSal"});$   
Projection  $P_2 = \text{Projections} \cdot \text{sum}(\text{"customerSal"});$

Projection  $P_3 = \text{Projections. max("employeeSal")};$

-112

112 Projection List      `plist = Projections.projectionList();`

plist.add(p1);

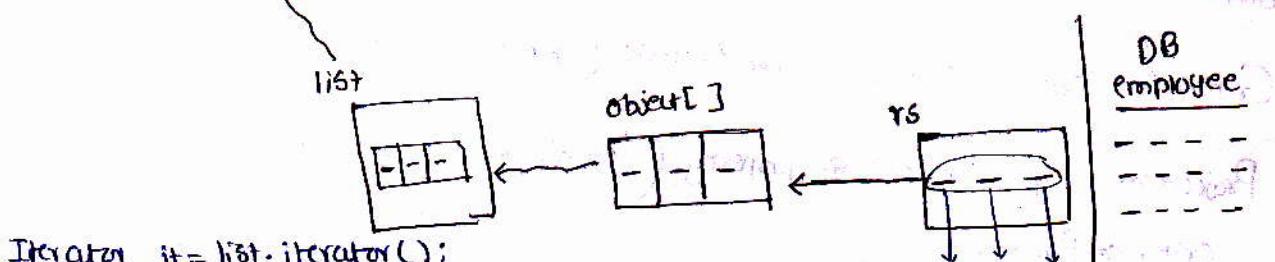
plist.add( $P_2$ );

plist.add( $P_3$ );

113

```
crit.serprojection(plist);
```

List list = crit.list();



Iterator it = list.iterator();

if (it.hasNext())

5

~~Object~~ raw[] = (Object[]) it.next();

## Pagination methods in Criteria :-

- While loading entities of a POJO class using Criteria, it is possible to load the objects of a particular range, instead of loading all objects from a DataBase.
- Reading some part of data from a huge amount of data is called applying pagination technique.

The following are the two methods used for pagination in Criteria

- 1) `setFirstResult(15)` → it starts from 16<sup>th</sup> record onwards to read the data
- 2) `setMaxResults(5)` → 16<sup>th</sup> record to 20<sup>th</sup> records to read bcz max result is 5

For Eg:-

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

`crit.setFirstResult(5);` reads the details of Employees for 6, 7 & 8 records

```
crit.setMaxResults(3);
```

```
List list = crit.list();
```

```
Iterator it = list.iterator();
```

```
while(it.hasNext())
```

```
{
```

```
Employee e = (Employee) it.next();
```

```
S.O.P(e.getEmployeeId() + " " + e.getEmployeeName() + " " + e.getEmployeeSal() + " " + e.getDeptNumber());
```

```
}
```

**CritTest**

→ Employee.java

employee.hbm.xml

hibernate.cfg.xml

CritClient.java

\*.class

refer page NO: 15 appn 10

Q:

choose HQL when the Queries are static, non-select operations

choose Criteria when the Queries are dynamic Queries, select operations on dynamic

## NativeSQL

Q:- When to choose HQL and When to choose Criteria in Hibernate?

Ans:- When we want to perform a Select operation and when we want to

apply multiple conditions i.e. we want to make the Query as dynamic

then it is recommended to use Criteria.

→ When we want to perform non-select operations (or) when we want to

execute static Queries then it is recommended to use HQL.

13/08/2013

→ NativeSQL is a concept of Hibernate, through which, we can execute direct SQL operations on a Database.

→ with NativeSQL, we can perform both Select and non-select operations on a Database.

→ In NativeSQL, we directly type SQL Queries, so NativeSQL is a Database dependent technique.

→ The following are the two major reasons for using NativeSQL in an Hibernate application.

① While migrating an existing JDBC application into an Hibernate appn, we use NativeSQL. Because a JDBC application contains SQL Commands and NativeSQL also allows SQL Commands.

② In order to call a "procedure" of a Database from an Hibernate application, we need NativeSQL.

→ For executing SQL Queries on a Database from an Hibernate application, we need a reference of org.hibernate.SQLQuery interface.

→ We can obtain a reference of SQL Query Interface, by calling createSQLQuery() method of SessionInterface.

Syntax:-

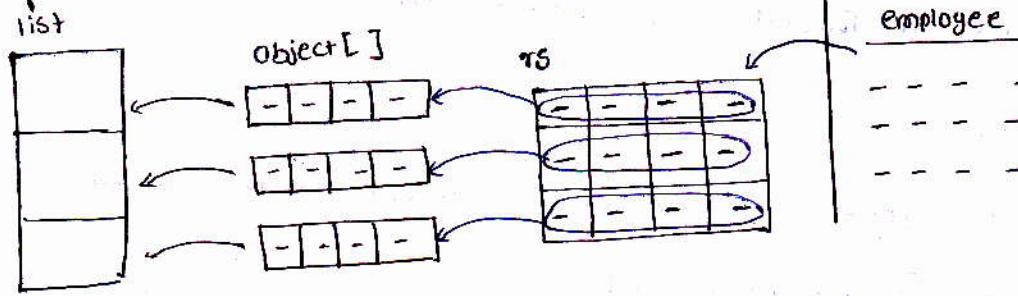
```
SQLQuery qry = session.createSQLQuery("sql command");
```

→ While selecting a Full Objects (or) a Full Object From a DataBase using NativeSQL, internally Hibernate Converts each row of ResultSet into an Object[], but not into a POJO class object. because in NativeSQL, we type directly SQL command and it does not contain a POJO class name in it.

For example:-

```
SQLQuery qry = session.createSQLQuery("select * from employee");
```

```
List list = qry.list();
```



```
Iterator it = list.iterator();
```

```
while (it.hasNext())
```

```
{
```

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

```
S.o.P(row[0] + " " + row[1] + " " + row[2] + " " + row[3]);
```

```
}
```

→ While executing a NativeSQL query for loading a Full Object (or) objects from a DataBase, then we can inform the hibernate that convert each row of

Resultset into a POJO class object, instead of an Object[]

→ To inform the hibernate that convert internally each row into a POJO class object, we call addEntity() method. To this method we pass class object of a POJO class as a parameter.

For example:-

```
SQLQuery qry = session.createSQLQuery("select * from employee");
```

```
qry.addEntity ( Employee.class );
```

```
List list = qry.list();
```

```
Iterator it = list.iterator();
```

```
while ( it.hasNext() )
```

```
{
```

```
Employee e = (Employee) it.next();
```

```
S.o.p( e.getXXX() );
```

```
}
```

### Executing a non-select operation :-

- We call executeUpdate() for executing a non-select operation on a DataBase
- While executing a non-select operation, we begin a Transaction. Because, in Hibernate a Transaction is required for executing non-select operations.

### For example:-

```
SQLQuery qry = session.createSQLQuery ("update employee set sal=? where  
empno=?");  
qry.setParameter ( 0, 8000 );  
qry.setParameter ( 1, 7188 );
```

```
Transaction tx = session.beginTransaction();
```

```
int k = qry.executeUpdate();
```

```
tx.commit();
```

### NativeSqlTest

```
Employee.java
```

```
employee.hbm.xml
```

```
hibernate.cfg.xml
```

```
NativeClient.java
```

```
*.class
```

refer page(16) appn (11) of handout.

Q:- What is the difference b/w HQL and NativeSQL?

- Ans:- ① In case of HQL, we use HQL queries, but in NativeSQL we use SQL queries.  
② HQL Queries are Database Independent, but SQL Queries are Database dependent.  
③ Through HQL, we can't call a procedure or a function of "Database", using NativeSQL we can call a procedure or a function of a Database.

### Calling a procedure or a Function :-

- In Hibernate, we have a limitation for calling a procedure or a function.
- The limitation is, a procedure can be called from Hibernate, if "it does not return any values". It means a procedure should not contain either out (or) inout parameters.
- We can call a Function of a DataBase from Hibernate, only if that function is returning a Cursor."
- While working with either JDBC or hibernate, it is not possible to create a Procedure or a function in DataBase from a Java program. It is only possible to call a procedure or a function from a Java program.

14/03/2012

- To call a procedure or a function when it is returning any no. of values or any type of values then we can use JDBC code in Hibernate for calling such type of procedures (or) functions

Calling a procedure (or) a function from Hibernate can be done in the following 2 ways.

- ① Using Hibernate without a JDBC code explicitly.
- ② Using Hibernate with JDBC code explicitly.

## Calling a Procedure from Hibernate:-

- If u want to call a procedure from Hibernate by without using JDBC code then that procedure should not return any values.
- We can call a procedure by constructing a reference of SQLQuery like the following  
SQLQuery qry = session.createSQLQuery("{call proname(Params)}");

The following procedure in Oracle takes two ~~parameters~~ and updates the given salary for an employee.

SQL> ed file1

```
in      in
create or replace procedure testpro(eno    number, salary number)
is
begin
  update employee set sal=salary where empno=eno;
end;
```

SQL> @x1  
→ compiling of procedure

SQL> @x1

Thisline  
toexecute

Procedure created

The following Hibernate application is for calling the above procedure.

Procedure Call

↳ hibernate.cfg.xml → mapping resource tag is removed from this file.  
client.java  
\*.class

// Client.java

```
import org.hibernate.*;
import org.hibernate.cfg.*;
```

```
public class Client {
```

```
  public void main(String[] args) {
```

```
  }
```

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

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

```
  SQLQuery qry = session.createSQLQuery("{call testpro(?,?)}");
```

```
qry.setInt(0, 7788);  
qry.setInt(1, 8500);
```

```
Transaction tx = session.beginTransaction();  
qry.executeUpdate();  
tx.commit();  
session.close();  
factory.close();  
System.out.println("procedure executed");
```

{

{

### Calling a Function from Hibernate :-

- To call a Function from Hibernate by without using JDBC code then the following 3 conditions should be satisfied.
  - ① Function must return a cursor of Database.
  - ② The cursor must return (or) must contain one or more rows.
  - ③ That function must call Named Queries concept.
- Named Query :-
  - While Developing an Hibernate application, if same HQL Query (or) SQL Query is required for multiple times then instead of retyping the same query again and again, we can use Named Queries Concept of Hibernate.
  - Named Queries are two types
    - ① Named HQL query
    - ② Native SQL query.
  - In order to reuse a query for multiple times in an Hibernate application then we need to configure that query in mapping file, by putting some name for the query.
  - To configure HQL query in a mapping file then the tag is <query>. Similarly, to configure SQLQuery the tag is <sql-query>

NOTE:- The `<query>` or `<sql-query>` tag should be added <sup>at</sup> outside of `<class>` tag.

For example :-

### Employee-hbm.xml

`<h-m>`

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

`....`

`....`

`....`

`</class>`

`<query name="q1">`

`from Employee e where e.deptNumber = ?` → HQL query

`</query>`

`<sql-query name="q2">`

`Select * from employee where sal > ?` → SQL query

`</sql-query>`

`</h-m>`

→ To execute either <sup>a</sup> Named HQL query (or) a Named SQL query, we need to obtain Query reference by calling `getNamedQuery()` method of Session Interface.

Query qry1 = session.getNamedQuery("q1"); → HQL query

Query qry2 = session.getNamedQuery("q2"); → SQL query

qry1.setParameter(0, 20); → deptnumber

List list = qry1.list();

↳ Employee pojo class object bcz Full object

qry2.setParameter(0, 2000);

List list = qry2.list();

↳ Object[] objects bcz it is SQL query

The following function is going to return a cursor with a set of Employees

SQL> ed x2

```
create or replace function emp_fun
return sys_refcursor is
  c1 sys_refcursor;
begin
  open c1 for select * from employee;
  return c1;
end;
```

drop function emp\_fun;

SQL> @x2

/

Function created

→ For calling the above function from Hibernate, we need to configure a NamedQuery

SQL query in a mapping file like the following

```
<sql-query name="q1" callable="true">
  <return class="Employee" alias="e" />
  {? = call emp_info(?)}
```

</sql-query>

Function

↳ employee.hbm.xml

hibernate.cfg.xml

Employee.java

client.java

refer page no. (17) in Handout  
appn (12)

In the above, <return> tag is used for informing the hibernate to convert

each row of Resultset into POJO class object.

→ If we do not configure <return> tag then the hibernate converts each row of

Resultset into an Object[].

→ In a client application of hibernate the following code is required for calling

which is the function, configured in the mapping file

```
Query qry = session.getNamedQuery("q1");
```

```
List list = qry.list();
```

---

→ when we call list() method then the following steps are executed.

① Hibernate calls the function and obtains a cursor object from it.

② Hibernate stores the cursor data into a Resultset object of JDBC

③ Hibernate stores the data of each row of Resultset into POJO class object.

④ Hibernate stores the POJO class objects into a collection of type List.

and finally hibernate ~~will~~ returns the Collection object back to a Java program.

Calling a procedure/ Function in Hibernate using JDBC code :-

→ Session interface of hibernate has provided a method doWork(), which is used to ~~execute~~ (or) execute the JDBC Code written in an Hibernate application.

→ In Hibernate, to write JDBC code then we need to implement org.hibernate.jdbc.Work interface

→ Work interface has provided an Abstract method called execute() and we provide implementation for execute() method by writing JDBC code.

→ For example:-

```
public class MyWork implements Work
{
    public void execute(Connection con) throws Exception
    {
        //Write jdbc code
    }
}
```

→ While calling procedures and functions from hibernate using jdbc code then there are no restrictions <sup>calling</sup> ~~written~~ the procedures and functions.

→ in execute() method of Work interface, we use CallableStatement of JDBC for calling a procedure (or) a function.

The following procedure is for calculating bonus by depending on salary of an Employee. This procedure takes empno has input then returns bonus as output.

SQL> ed <sup>filename</sup> temp2

Create or replace procedure bonus\_Pro(eno in number, bonus out number)

is  
empSal number(9,2);

begin <sup>to create variable</sup>

Select sal into empSal from employee where empno = eno;

if empsal between 0 and 2000 then

    bonus := 500;

elseif empsal between 2001 and 5000 then

    bonus := 1000;

else

    bonus := 1500;

end if;

end;

SQL > @tmp2

/  
procedure created

### ProcedureCall

↳ hibernate.cfg.xml

Client.java / .class

// Client.java

import org.hibernate.\*;

import org.hibernate.cfg.\*;

import org.hibernate.jdbc.\*;

import java.sql.\*;

import java.util.\*;

public class Client

{ public static void main(String args[])

{

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

    Session session = factory.openSession();

    Scanner s = new Scanner(System.in); → To give input for runtime dynamically

    System.out.println("Enter Employee ID");

    s.nextInt();

In anonymous class

we inside use  
eno in anonymous class

so, we define final

    final int eno = s.nextInt();

    session.doWork(new Work())

    { → throw self implemented exception → SQLException or  
        public void execute(Connection con) throws SQLException

        { → nothing to do → nothing to do → to form set

CallableStatement cstmt = com.prepareStatement("call bonus\_pro(?,?)");

```

    //register Out parameter
    Cstmt.registerOutParameter(2, Types.INTEGER);

    //set in parameter
    Cstmt.setInt(1, eno);
    //call the Procedure
    Cstmt.execute();

    int bonus = Cstmt.getInt(2);
    S.o.p("Bonus=" + bonus);
}

//end of execute()

//end of Work implementation
);
//end of doWork()
Session.close();
Factory.close();
}
}

```

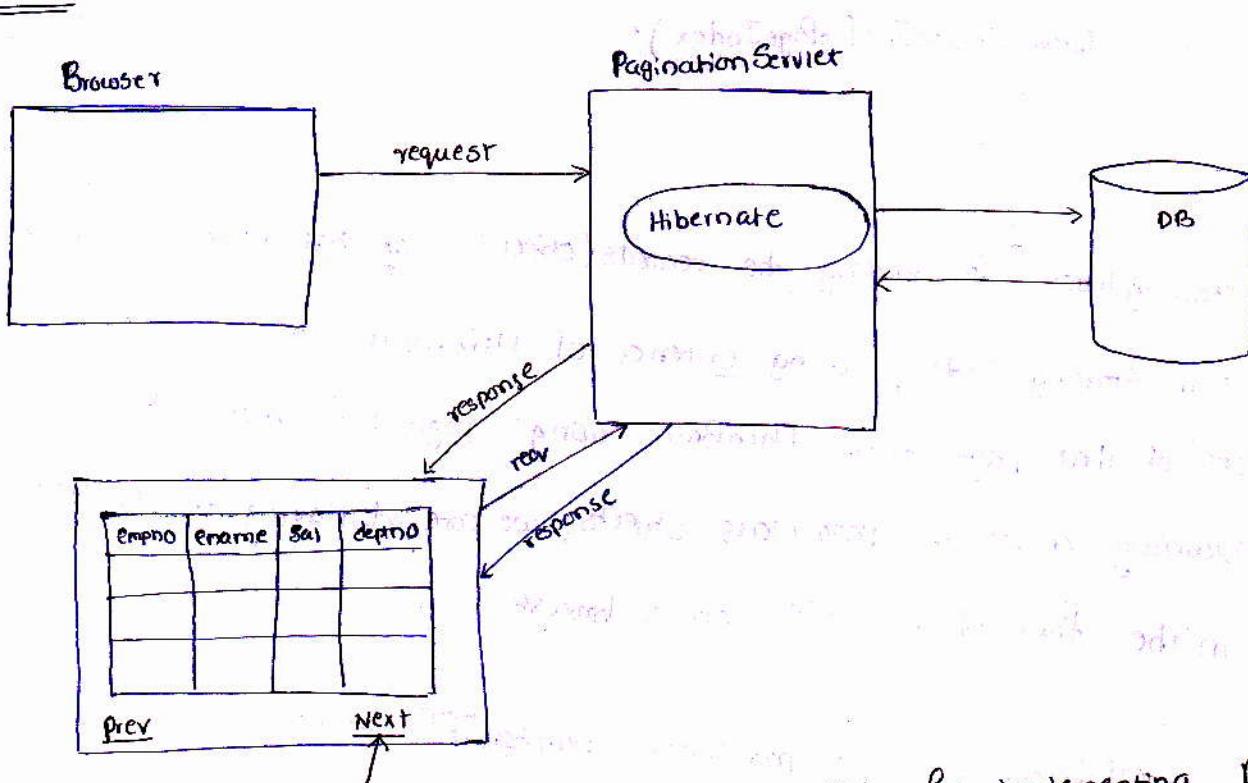
### Pagination with Servlet and Hibernate Integration :-

17/03/2013

- When there is a huge amount of data to be displayed on a browser, as part of a response for a given request then, if all the data is displayed on the browser then an end user loses concentration, while reading the huge amount of data at once, shown on a browser.
- In order to overcome the above problem, we need divided the huge amount of data into multiple pages and we need to show page by page of data on a browser. This mechanism is called "pagination".
- For example, if gmail, all the mails of Inbox are not displayed at a time on the browser. The mails are displayed in a page by page. It means pagination mechanism is applied for displaying the mails of an Inbox.
- In most of web applications used in a realtime, (a) applied with pagination

- The following example is a Servlet with Hibernate Integration, to display 3 Employee Objects details at once in a page, by deviding the Employees plages into multiple pages using pagination mechanism.

### Flow:



- The following logics are required in a servlet for implementing Pagination mechanism.

### logic 1:-

- In a servlet, we need to check whether a request is made for which page of response data.
- In a servlet, we are reading the value of a request parameter called pageIndex, to find the page requested by the client.
- we are deciding pageIndex as one, if a client is not passing pageIndex parameter.
- If passed pageIndex then we are taking the value of pageIndex parameter as a pageIndex in our servlet.

```

String sPageIndex = request.getParameter("pageIndex");
if(sPageIndex == null)
{
    pageIndex = 1;
}
else
{
    pageIndex = Integer.parseInt(sPageIndex);
}

```

### logic2:-

- Find the startingIndex for reading the records(objects) of that page from a DataBase.
- After finding Starting Index, using Criteria of Hibernate we are reading the objects of that page from DataBase using pagination methods.
- After obtaining a list of pojo class Objects, we are displaying the employees details in the form of a table on a browser.

```

int si = numberOfRecordsPerPage * pageIndex - numberOfRecordsPerPage

```

```

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

```

```

crit.setFirstResult(si);

```

```

crit.setMaxResults(numberOfRecordsPerPage);

```

```

List list = crit.list();

```

```

Iterator it = list.iterator();

```

```

while(it.hasNext())
{
    .....
}

```

### logic3:-

- Find total no of objects (records) available in the DataBase table.
- In hibernate, while using Criteria, we need projections for finding an aggregate function result.
- In this logic, we are storing the no of objects available in DataBase in totalnoofrecords Variable.

```

Criteria crit = session.createCriteria(Employee.class);
Projection p1 = Projections.rowCount();
crit.setProjection(p1);
List list = crit.list();
Iterator it = list.iterator();
if(it.hasNext())
{
    Integer i = (Integer)it.next();
    totalNumberOfRecords = i;
}

```

#### Logic 4:-

- Find no.of pages required for displaying the total no.of records
- we devide the total no.of records with no.of records per page for finding the no.of pages
- If any additional records are exist then increment the no.of pages by one.

```

noOfPages = totalNumberOfRecords / numberOfRecordsPerPage;
if(totalNumberOfRecords > (noOfPages * numberOfRecordsPerPage))
{
    noOfPages++;
}

```

#### Logic 5:-

- Prev and Next links hyperlinks on browser for PageNavigation.
- pass a parameter pageIndex to a servlet when the link is linked
- for previous link, pageIndex is  $currentPageIndex - 1$  and for Next Link the pageIndex is  $currentPageIndex + 1$ .
- Previous link should not be displayed on FirstPage and Next Link should not be displayed on LastPage.

```

if(pageIndex > 1)
{
    pw.println("<a href=srv1?pageIndex=" + (pageIndex - 1) + ">Prev</a>");
}

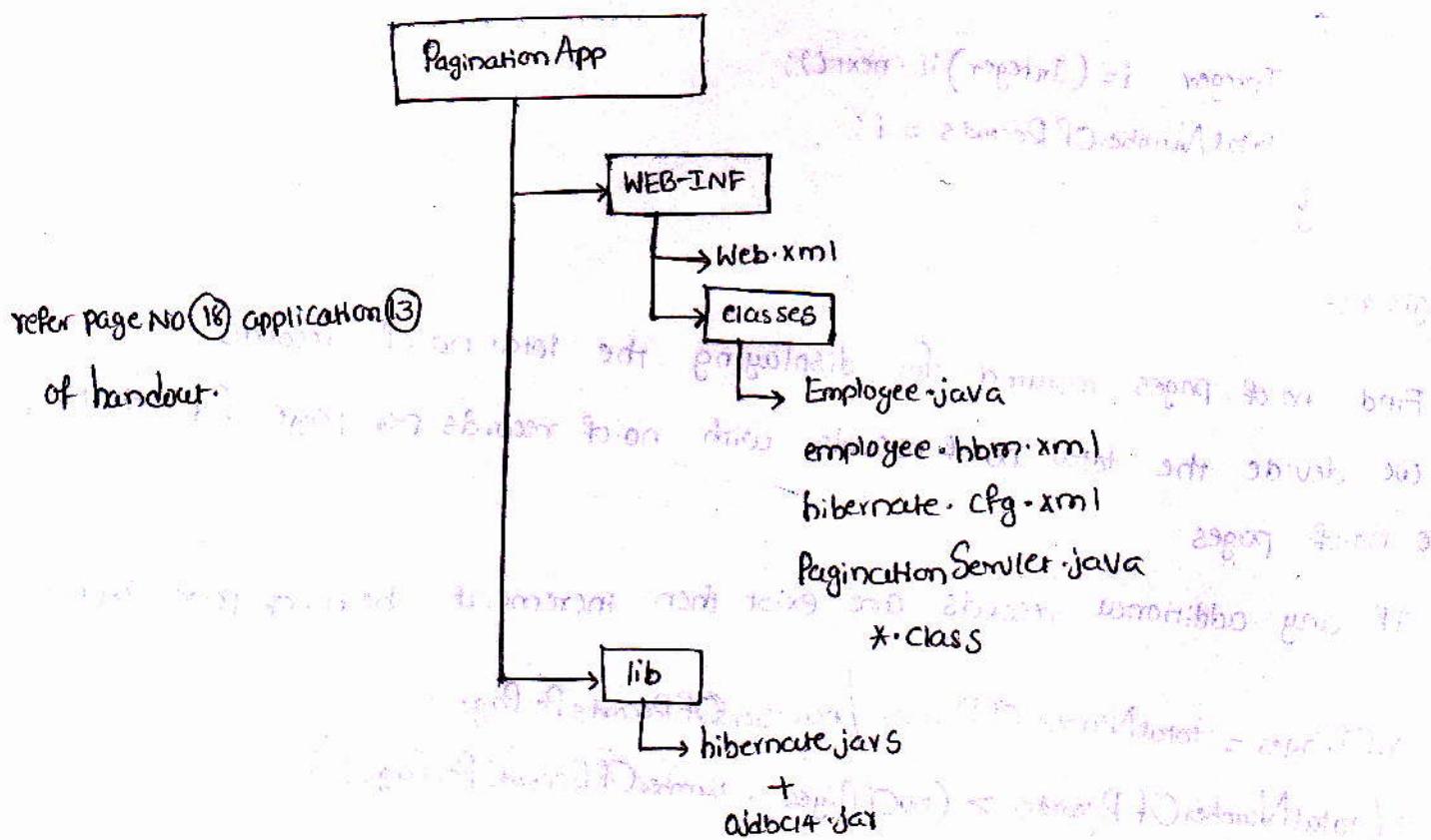
```

```

if(pageIndex < noOfPages)
{
    pw.println("<a href=srv1?pageindex=" + (pageIndex + 1) + ">Next</a>");
}

```

Directory structure :-



To compile pagination Servlet, we need to set the following two jar files in the CLASSPATH

- ① servlet-api.jar
- ② hibernate3.jar

request:

http://localhost:2020/PaginationApp/ps

## Relationships in Hibernate (Associations in Hibernate)

→ The data in a DataBase is stored in multiple tables and then some relation is assigned b/w the Data, because of the following two reasons.

- ① In order to reduce ~~Data~~ Redundancy of the Data in DataBase tables
- ② To manage the operations on the data easily.

for Example:- we have a table Hospital, where we want to store the Doctors and patients taken appointment for a doctor. so, the table is going to look like the following.

Hospital			Pat-ID	Pat-Name	Timeslot	
DocID	DocName	Specialization				
101	ABC	MD	1	X	9:00	
	ABC	MD	2	Y	6:15	
	-	-	3	-	-	
	-	-	4	-	-	
	-	-	5	-	-	
Duplicate			301	A	7:15	
Duplicate				B	7:20	
-				-	-	
-				-	-	
-				-	-	

In the above table,

a doctor information is repeated for each patient information.  
so, a Doctor information is duplicated in the table for multiple times.  
so, it leads to DataRedundancy.

→ To solve the above DataRedundancy problem, we can devide the above data table into two tables. like the following

doctor-info		
DocID	DocName	Specialization
101	ABC	MD
102	XYZ	MS

patient-info			Timeslot
Pat-ID	Pat-Name		
1	X		
2	Y		
301	A		
302	B		

In the above, the data is divided into two tables, there is no relation b/w the data in two tables. So, the data stored in the DataBase is an Useless data.

→ So, when the data is divided across two tables then there should be some relation b/w the data, to make it as useful data.

→ In the above, we have two options, to make the data as usefull.

① We can copy a patient id into Doctor-info table.

② We can copy a doctor id into Patient-info table.

→ in the 1<sup>st</sup> case, for each Patient id, again a Doctor information is repeated. it means still there is a DataRedundancy.

doctor-info

<u>docid</u>	<u>docname</u>	<u>specialization</u>	<u>Pat-id</u>
101	ABC	MD	1
101	ABC	MD	2
101	ABC	MD	3
101	ABC	MD	4
101	ABC	MD	5
101	ABC	MD	6
101	ABC	MD	7

/duplicated.

→ To resolve this DataRedundancy problem, we need to Copy the Doctor <sup>id</sup> into patient-info table. so that for each patient information, only a Doctor id is repeated. so, the Redundency problem is reduced.

Patient-info

<u>Pat-id</u>	<u>Pat-name</u>	<u>timeslot</u>	<u>doc_id</u>
1	x	---	101
2	y	----	101
3	z	---	101
301	A	---	102
302	B	----	102

finally the Redundency problem is reduced by copying a doctorid into a Patient information. So, According to the DataBase terminology, a referential Integrity is applied b/w the data of two tables.

- According to DataBase terminology, doctorid is copied to patient, so doc-info is a parent table (or) master table and Patient-info table is a child table (or) Detailed table.
- in Hibernate
- While working with hibernate for persisting this type of related data in DataBase, we create multiple POJO classes as Part of our application. And we assign a relationship b/w Objects of the POJO classes. This concept is called Association mapping in hibernate.
- Hibernate supports the following 4 types of relationships b/w objects.

- ① one-to-many
- ② many-to-one
- ③ many-to-many
- ④ one-to-one

### One-to-many (Uni-directional) :-

- relationship
- To apply this one-to-many association between the objects of two POJO classes in Hibernate, then the following changes are required
  - ① while creating a parent POJO class, we need to add an additional property of type collection. This collection property is used to store many child objects in a Parent object.
  - ② we need to configure the Collection Property in a parent class mapping. During this configuration, we need to configure the Foreign key and the relationship in the mapping file.
  - ③ In a client application of hibernate, child objects are added to the parent like the following
    - a) add a group of child objects to a collection object
    - b) add collection object to the parent object

Example:-

- We have two POJO classes in application called Vendor and Customer.
- The relationship between these two class objects is, <sup>one</sup> Vendor is associated to multiple customers. so the relationship is one-to-many.
- In this Vendor and Customer relationship, Vendor class is a parent class and Customer class is a child class.
- The One-to-many relation between Vendor to Customer is going to use the following two tables in DataBase.

<b>PK ↓</b>	VENDOR (parent)	
<b>VID</b>	<b>VNAME</b>	

//Parent

Vendor.java

public class Vendor

```
{
    private int vendorId;
    private String vendorName;
}
```

① private Set customers;

setters & getters

}

Vendor.hbm.xml

L1-M

```
<class name="Vendor" table="vendor">
    <id ... />
    <property ... />
```

② <set name="customers">

<key column="VENID" />

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

<set>

<class>

</h-m>

<b>PK ↓</b>	CUSTOMER (child)		<b>FK ↓ VEN VID</b>
<b>CUSTID</b>	<b>CUSTNAME</b>	<b>CUSTADDR</b>	

//child

Customer.java

public class Customer

{

```

    private int customerId;
    private String customerName;
    private String customerAddress;
```

setters & getters

}

Customer.hbm.xml

L1-M

<class name="Customer" table="Customer">

<id ... />

<property ... />

<property ... />

<class>

## // Client.java

```
public class Client  
{  
    public static void main(String[] args)  
    {  
        // ...  
    }
```

③ a) Set set=new HashSet();

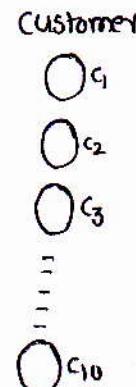
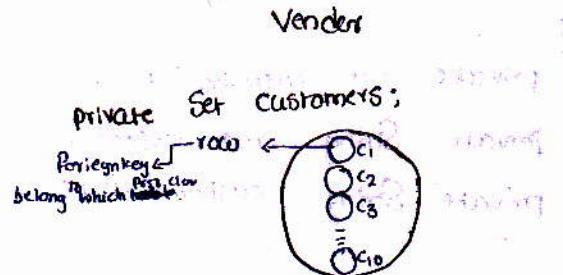
```
    set.add(c1);  
    set.add(c2);  
    set.add(c3);
```

b) v.setCustomers(set);

```
Transaction tx=session.beginTransaction();  
session.save(v);  
tx.commit();
```

}

Method overriden  
parent class save method



NOTE:- While working with relationships of hibernate, in order to inform the hibernate that apply the operation done on one object on its relationship objects also, we need to add an attribute called **Cascade**, while configuring the collection element in a mapping file.

for Eg:-

```
<set name="customers" cascade="all">  
    <key column="venid"/>  
    <one-to-many class="Customer"/>  
</set>
```

If we don't add cascade attribute then the default value of cascade is **None**.

It means, by default hibernate doesn't transfer any operations on to the related objects.

→ In a DataBase, the vendor and customer tables can be created for this one-to-many relationship like the following.

```
sql> Create table vendor (vid number(5) primary key, vname varchar2(20));
```

```
sql> Create table customer (custid number(5) primary key, custname varchar2(10),  
    custaddr varchar2(10), vendid number(5) references vendor (vid));
```

→

### OneToMany

↳ vendor.java

Customer.java

Vendor.hbm.xml

Customer.hbm.xml

hibernate.cfg.xml

InsertClient.java.

\*.class

//ParentPOJO

//Vendor.java

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

public class Vendor

{

private int vendorId;

private String vendorName;

private Set customers;

=

=

}

//Customer.java

```
public class Customer
```

{

private int customerId;

private String customerName;

private String customerAddress;

=

=

}

```
<!-- vendor.hbm.xml -->
```

```
<h-m>
```

```
  <class name="Vendor" table="vendor">
```

```
    <id name="vendorId" column="vid"/>
```

```
    <property name="vendorName" column="vname" length="10"/>
```

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

```
      <key column="venid"/>
```

```
      <one-to-many class="Customer"/>
```

```
    </set>
```

```
  </class>
```

```
</h-m>
```

```
<customer.hbm.xml -->
<h-m>
<class name="Customer" table="Customer">
  <id name="customerId" column="custId"/>
  <property name="customerName" column="custName" length="10"/>
  <property name="CustomerAddress" column="custAddress" length="10"/>
```

```
</class>
```

```
</h-m>
```

```
21--hibernate.cfg.xml -->
```

```
=
=
=
<mapping resource="Vendor.hbm.xml"/>
<mapping resource="Customer.hbm.xml"/>
```

```
// InsertClient.java
```

```
import java.util.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
public class InsertClient
{
  public static void main(String[] args)
  {
    SessionFactory factory = new
    Session session = factory.openSession();
    //Parent Object
    Vendor v = new Vendor();
    v.setVendorId(111);
    v.setVendorName("IBM");
  }
}
```

```
// child Object - 1
```

```
Customer c1 = new Customer();
c1.setCustomerId(501);
c1.setCustomerName("INFY");
c1.setCustomerAddress("HYD");
```

```
// child Object - 2
```

```
Customer c2 = new Customer();
c2.setCustomerId(502);
c2.setCustomerName("TCS");
c2.setCustomerAddress("HYD");
```

// child Object-3

```
Customer c3 = new Customer();  
c3.setCustomerID(503);
```

```
c3.setCustomerName("VERIZON");
```

```
c3.setCustomerAddress("US")
```

// add child Objects to Java.util.Set

```
Set s = new HashSet();
```

```
s.add(c1);
```

```
s.add(c2);
```

```
s.add(c3)
```

// add java.util.set to Parent Object

```
v.setCustomers(s);
```

Transaction tx = session.beginTransaction();

```
session.save(v);
```

```
tx.commit();
```

```
Session.close();
```

```
factory.close();
```

```
}
```

```
y
```

SQL> Select \* from Vendor;

VENDID	VNAME
111	IBM

SQL> Select \* from customer;

CUSTID	CUSTNAME	CUSTADDR	VENDID
501	INFY	HYD	111
502	TCS	HYD	111
503	VERIZON	US	111

## Select Operation in One-to-many :-

- In one-to-many relationship, when a parent Object is loaded hibernate automatically loads its child objects <sup>are</sup> also from the DataBase.
- While loading the child objects of a From Parent Object, we can inform the hibernate whether Lazy loading (or) early loading of child objects is required through "lazy" attribute.
- By default hibernate uses lazyloading, because the default value of "lazy" attribute is "true".
- If we want to inform the hibernate that, apply earlyloading on child objects then we should make the value of "lazy" attribute as "false"
- In early loading, hibernate loads the child Objects along with the parent Object immediatly. But in Lazyloading, first only parent Object is loaded and its child objects are loaded on demand (when ever required)

```
<set name="customers" cascade="all" lazy="true/false">  
    ↓  
    ↓  
    ↓
```

```
</set>
```

lazy = "true" → lazyloading  
lazy = "false" → earlyloading

```

    // Step-1
    Vendor v = (Vendor) session.get(Vendor.class, 111);

    // Step-2
    Set s = v.getCustomers();

    // Step-3
    Customer c = (Customer) session.get(Customer.class, 503);

    // Step-4
    Transaction tx = session.beginTransaction();
    s.remove(c);
    tx.commit();
  
```

- When a child object is removed from a collection of child objects of a parent object then hibernate is going to remove the relationship between the parent object and the child object, by making Foreign key of that child object as "null"
- After a relationship is removed b/w a parent and a child object, hibernate is not going to remove that record from the DataBase.
- If the above client application is executed then in Customer table of DataBase a Customer record with "503" existing table with Foreign key as "null".

<u>Customer</u>				
<u>Custid</u>	<u>custname</u>	<u>custaddr</u>	<u>venid</u>	
501	INFY	HYD	111	
502	TCS	HYD	111	
503	VERIZON	US	111	Orphan record.
504	CTS	HYD	111	

- Even though the relationship is removed, but still the record is exist in the table. We call such record as a "Orphan record".
- If we want to inform the Hibernate that remove orphan record from the DataBase table immediately, whenever the relationship is removed b/w the parent and child record then we need to modify a value of cascade attribute

as "all-delete-orphan".

```
<set name="customers" lazy="true" cascade="all-delete-orphan">
```

```
</set>
```

→ for cascade attribute, we have the following 6 values.

- 1) None (default)
- 2) all
- 3) all-delete-orphan
- 4) save
- 5) save-update
- 6) delete

→ Cascade attribute is used to inform the hibernate about which non-select operations should be transferred (applied (cascaded)) on relationship object of an object.

→ If "cascade = save" then Hibernate only cascades "insert" operation on child objects.  
If "cascade = save-update" then Hibernate cascades "insert" and "update" operations on child objects.  
If "cascade = delete" then Hibernate cascades only "delete" operation on child objects.  
If "cascade = all" then Hibernate cascades insert, delete, update operations on child objects.  
If "cascade = all-delete-orphan" then Hibernate cascades all operations on child objects and also deletes orphan records.

→ In one to many relation, if we want to remove the all child objects of a parent object then follow the below steps.

- ① load the parent object from the DataBase
- ② read all child objects ~~one~~ of a parent object into collection
- ③ remove all objects from a collection with in a Transaction

//Step-1

```
Vendor v = (Vendor) session.get(Vendor.class, 111);
```

//Step-2

```
Set s = v.getCustomers();
```

//Step-3

```
Transaction tx = session.beginTransaction();
```

```
s.removeAll();
```

```
tx.commit();
```

→ If you want to remove a parent object along with its all child objects from the DataBase then we need to delete simply the parent object. like the following

```
Vendor v = (Vendor) session.get(Vendor.class, 11);
Transaction tx = session.beginTransaction();
session.delete(v);
tx.commit();
```

Using `java.util.List` as a collection :-

→ When applying one-to-many relationship, for storing a group of child objects to a parent object, in Parent class we need a collection of type either `Set` (`java.util.Set`) or `List` or `Map`.

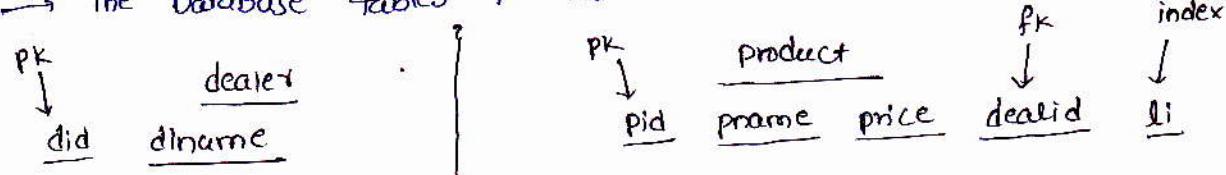
→ If we use collection type as `List` then hibernate uses an additional column in child table for storing "index" of a child in the List into DataBase.

→ At the time of Configuring `<list>` tag in a mapping file, we need to configure "listIndex" also in the mapping file.

Eg:- we have two POJO classes called Dealer and Product. if we are applying ~~one~~ one-to-many relation b/w a Dealer and a Product & i.e. one dealer is associated with multiple Products.

→ In Product table of DataBase (child table), an additional column is required for storing the "list index"

→ The DataBase tables for this Example are



OneToMany 2  
↳ Dealer.java  
Product.java  
Dealer.hbm.xml  
Product.hbm.xml  
hibernate.cfg.xml  
InsertClient.java  
\*.class

```

//parent
//Dealer.java
import java.util.*;
public class Dealer
{
    private int dealerId;
    private String dealerName;
    private List products;
}

setters & getters
}

```

```

//child
//Product.java
public class Product
{
    private int productId;
    private String productName;
    private double price;
}

setters & getters
}

```

### Dealer.hbm.xml

```

<h-m>
<class name="Dealer" table="dealer">
    <id name="dealerId" column="did"/>
    <property name="dealerName" column="dlname" length="10"/>
    <list name="products" cascade="all">
        <key column="dealid"/>
        <list-index column="li"/>
        <one-to-many class="Product"/>
    </list>
</class>

```

### product.hbm.xml

```

//InsertClient.java
import java.util.*;
import org.hibernate.*;
import org.hibernate.cfg.*;

```

```

public class InsertClient
{
    public void main(String[] args)
    {
        SessionFactory factory = new Configuration().configure().buildSessionFactory();
        Session session = factory.openSession();
        // Parent Object
        Dealer d = new Dealer();
        d.setDealerId(101);
        d.setDealerName("ABC");
        // child Object - 1
        Product p1 = new Product();
        p1.setProductId(901);
        p1.setProductName("X");
        p1.setPrice(2000);
        // Step-1 // add child Objects to java.util.List
        List list = new ArrayList();
        list.add(p1);
        list.add(p2);
        list.add(p3);
        // Step-2 // add java.util.List to parent
        d.setProducts(list);
        Transaction tx = session.beginTransaction();
        session.save(d);
        tx.commit();
        session.close();
        factory.close();
    }
}

```

Select \* from dealer;

DID	DLNAME
101	ABC

Select \* from product;

PID	PNAME	PRICE	DEALID	L1
901	X	2000	101	0
902	Y	3000	101	1
903	Z	4000	101	2

<bag> tag:-

- When a collection is used of type `java.util.List` then we configure `<List>` tag in a mapping file. If we configure `<list>` tag then we also need to configure `<list-index>` tag inside it, for informing the hibernate about the column name in child table, in which the index of the child object is need to be stored.
- When a child object is stored in DataBase then hibernate also stores its index in a list along with that child object in the DataBase. But there is no use with the index of a child object stored in a DataBase.
- In order to inform the hibernate that store only the child object, but not its index in a DataBase, we need to configure `<bag>` tag, instead of `<list>` tag in a hibernate mapping file.
- For the above example, in `dealer.hbm.xml` file, instead of `<list>` tag we can configure `<bag>` tag like the following.

```

<bag name="products" cascade="all">
    <key column="dealid" />
    <one-to-many class="product" />
</bag>

```

NOTE:- By configuring the `<bag>` tag, we can avoid an additional column in the child table (`list-index`), used for storing index.

Q:- What is the difference b/w using a collection of type `java.util.Set` and `java.util.List`?

Ans:- In case of `java.util.Set`, it is not possible to read a particular child object of the parent, by without loading all child objects from the DataBase.

→ In case of `java.util.List`, it is possible to read a particular child object for the parent by without loading all child objects.

(ii) because, List allows index based accessing of an Object.

(iii) The difference can be found, when Lazy loading is applied, but not in early loading.

For eg:- by using the following code, we can read only 4<sup>th</sup> child of the parent from the DataBase, by withholding all child objects from the DataBase.

\* ) Lazy = "true" → lazy loading

```
Dealer d=(Dealer) session.get(Dealer.class, 101);
```

```
List list=d.getProducts();
```

```
Object o=list.get(3);
```

```
Product p=(Product)o;
```

### Using collection type java.util.Map:-

→ While creating one-to-many relationship for storing a group of child objects in a parent Object, we use a collection type of property in a parent class.

That collection type can be either a java.util.Set (or) java.util.List (or) java.util.Map.

→ If we use a collection type as java.util.Map then in Mapping file of Hibernate, we should use <Map> tag

→ If collection type is used as a Map then, in child table of DataBase an additional column is required for an Hibernate, for storing the key assigned for a child Object, while storing it in Map.

→ If collection type is java.util.Map then, while configuring the <Map> tag, we need to configure an additional tag also inside <map> tag called <map-key>

java.util.Set → <set>

java.util.List → <list> / <bag> / <idbag>

java.util.Map → <map>

Eg: \* ) We have two pojo classes called Dealer and Product & the Relationship is a one-to-many from a dealer to a product.

\* ) In dealer pojo class, we are using "Map" type of collection for storing a group of products to a dealer.

\* ) At the time of inserting a Dealer with Products in Database then hibernate also stores the key of each product used in a "Map" collection in the DataBase

### One-to-many 3

→ Dealer.java  
Product.java  
Dealer.hbm.xml  
product.hbm.xml  
hibernate.cfg.xml  
InsertClient.java  
\*.class

// Dealer.java

```
import java.util.*;  
public class Dealer  
{  
    private int dealerId;  
    private String dealerName;  
    private Map products;  
    setters & getters.  
}
```

// Product.java

```
public class Product  
{  
    private int productId;  
    private String productName;  
    private double price;  
    setters & getters.
```

// dealer.hbm.xml

```
<!DOCTYPE hibernate-mapping>  
<h-m>  
    <class name="Dealer" table="dealer">  
        <id name="dealerId" column="did"/>  
        <property name="dealerName" column="dname" length="10"/>  
        <map name="products" cascade="all">  
            <key column="dealid"/>  
            <map-key column="mkey" type="java.lang.String" length="10"/>
```

One-to-many class="Product"/>

</map>

</class>

</h-m>

// product.hbm.xml

```
<mapping resource="product.hbm.xml"/>  
    <mapping resource="product.hbm.xml"/>
```

## // InsertClient.java

```
import java.util.*;  
import org.hibernate.*;  
import org.hibernate.cfg.*;
```

```
public class InsertClient
```

```
{
```

```
    public void (String [] args)
```

```
{
```

```
        SessionFactory factory =
```

```
        Session session =
```

```
// Parent Object
```

```
Dealer d = new Dealer();
```

```
d.setDealerId(101);
```

```
d.setDealerName("ABC");
```

```
// child object-1
```

```
Product p1 = new Product();
```

```
p1.setProductId(901);
```

```
p1.setProductName("X");
```

```
p1.setPrice(2000);
```

```
// child object-2
```

```
Product p2 = new Product();
```

```
p2.setProductId(902);
```

```
p2.setProductName("Y");
```

```
p2.setPrice(3000);
```

```
// child object-3
```

```
Product p3 = new Product();
```

```
p3.setProductId(903);
```

```
p3.setProductName("Z");
```

```
p3.setPrice(4000);
```

```
// step-1 // add child objects to Java.util.Map
```

```
Map map = new HashMap();
```

```
map.put("prod1", p1);
```

```
map.put("prod2", p2);
```

```
map.put("prod3", p3);
```

```
// step-2 // add java.util.Map to Parent Object
```

```
d.setProducts(map);
```

```
Transaction tx = session.beginTransaction();
```

```
session.save(d);
```

```
tx.commit();
```

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

```
factory.close();
```

```
Select * from dealer;
```

Did	Dname
101	ABC

```
Select * from product;
```

Pid	Pname	Price	DealId	Mkey
901	X	2000	101	Prod1
902	Y	3000	101	Prod2
903	Z	4000	101	Prod3

according to the hashcode map data structure will be stored data in reverse order.

Q:- What is the diff b/w using `java.util.List` and `java.util.Map`?

Ans:- In case of List, we can avoid the additional column in child table by Configuring `<Bag>` tag. But In case of Map, it is not possible to avoid additional column in child table.

- The Similarity b/w using a collection type as List and a collection type as Map is, it is possible to obtain a particular child object of the parent object, instead of loading all child objects of the parent object
- In case of collection type List, we use index and in case of collection type Map we use `<map-key>`

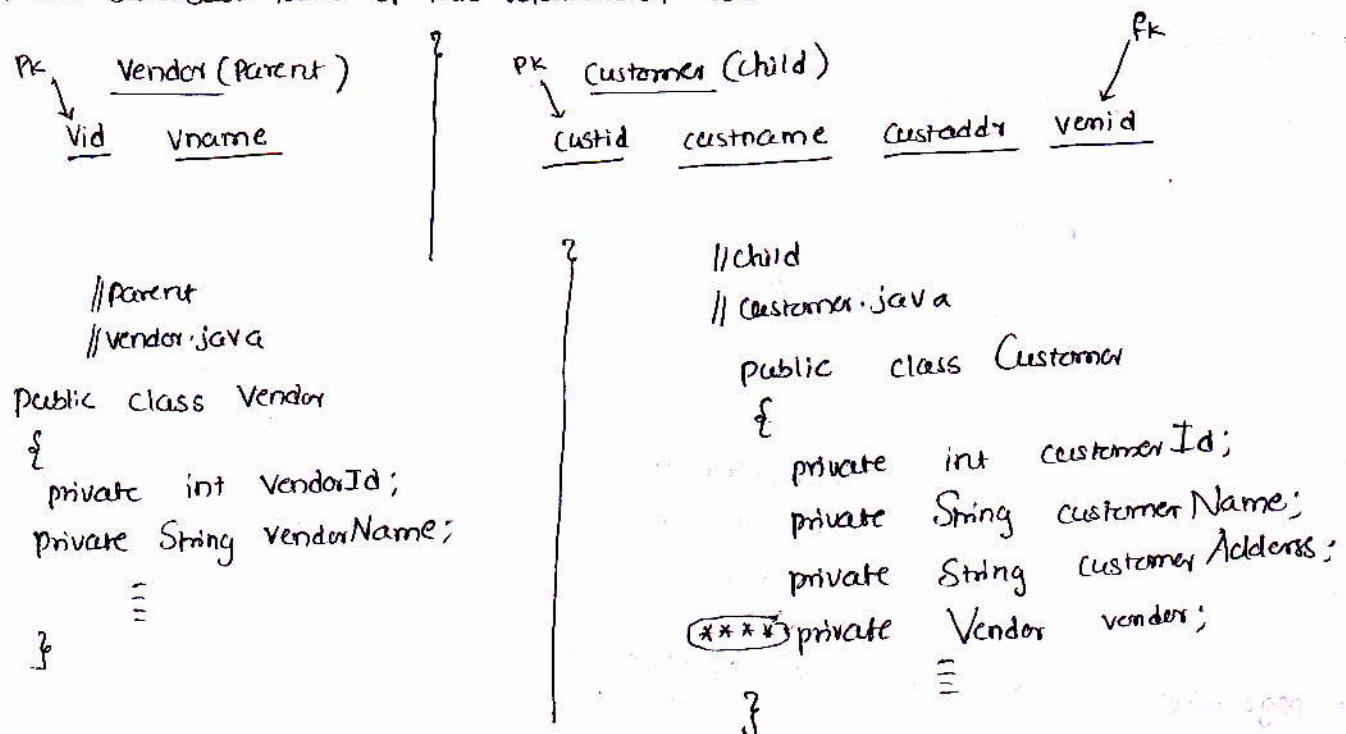
The following is to read a particular child of a parent object:-

```
Dealer d = (Dealer) session.get(Dealer.class, 111);
Map map = d.getProducts();
Object o = map.get("prod2");
Product p = (Product) o;
System.out.println(p);
```

## Many-to-one

- Many-to-one relationship can be applied from a child object to a parent object.
- According to Many-to-one, a parent object is added to many child objects.
- To apply Many-to-one association from a child object to its parent object we need the following changes.
  - ① in child pojo class, we need a reference, pointing to its parent class.
  - ② in child mapping file, we need to configure `<many-to-one>` tag, to inform the hibernate about the relationship.
- In hibernate, working with these relationships, we need a collection in a parent pojo class for to-many relation and we need a reference of parent class in a child class when the relation is to-one.
- Example:- we have two pojo classes Vendor and customer & the relation of many-to-one is a Vendor object is added to many Customer objects.

→ The DataBase table of this relationship are.



```

<h-m>
<class name="Vendor" table="vendor">
    <id --- />
    <property -- />
</h-m> <class>

```

//Customer.hbm.xml

<h-m>

<class name="Customer" table="customer">

<id -- />

<property -- />

<property -- />

<many-to-one name="vendor" class="Vendor" column="venid" cascade="all" />

↓  
reference of  
Vendor

↓  
parent class name

↓  
foreign key

</class>

<h-m>

→ in a client application of hibernate a Vendor object is added to the Customer Objects by calling setVendor() method of Customer POJO class.

Eg:-

C1.setVendor(v);

C2.setVendor(v);

C3.setVendor(v);

→ At the time of saving a Customer object, hibernate 1st verifies whether its Parent Object already exist in the database (or) not. If exist then only child object will be saved in the DataBase.

If parent object doesn't exist in the DataBase then hibernate 1st will save Parent object and then after the child object.

many-to-one

Vendor.java

Customer.java

Vendor.hbm.xml

Customer.hbm.xml

hibernate.cfg.xml

InsertClient.java

\*.class

Q1) select \* from vendor;

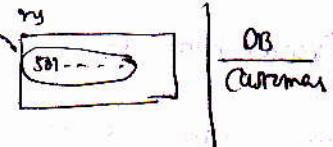
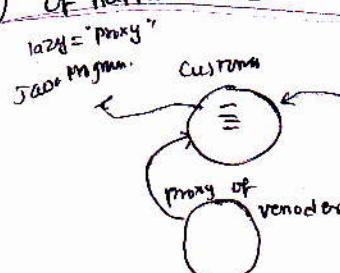
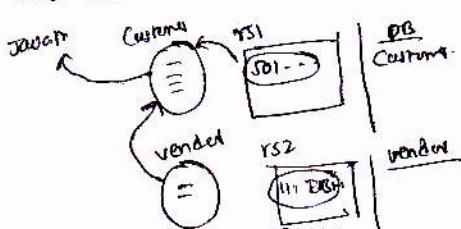
Vid	Vname
111	IBM

select \* from customer;

CustId	CustName	CustAdd	VENID
501	Infy	HYD	111
502	TCS	HYD	111
503	Verizon	US	111

refer page no (3) an application (15) of handout (2)

lazy="false"

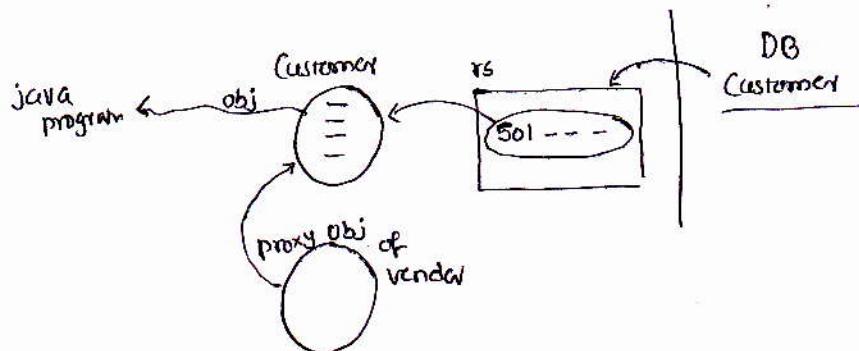


### "lazy" attribute of many-to-one :-

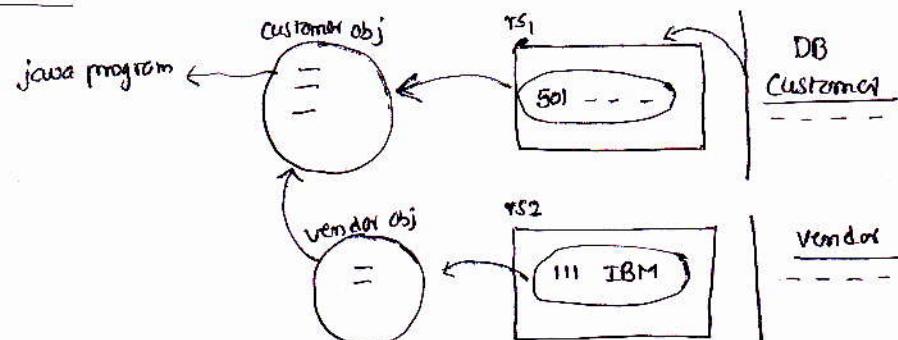
- in Many-to-one relationship, whenever a child object is loaded from the database then its associated parent object is also loaded automatically.
- in this Many-to-one, hibernate decides whether a parent object or a child to be loaded either immediately or later, by depending on "Lazy" attribute.
- The default value of lazy attribute in many-to-one is "proxy". This is called proxy loading.
- If you write "lazy=false" then it is called "early loading". In proxy loading, hibernate creates a proxy object of parent and then it is added to child object and after that the child object is given to our java program.
- In case of early loading, hibernate creates a real parent object and then it is added to the child object and finally the child object is given to java program.

Eg:- Customer c = (Customer) session.get(Customer.class, 501);

lazy="proxy" :-



lazy="false" diagram:-



## Delete operation in many-to-one :-

- When we are trying to delete a child object then hibernate tries to delete its parent object also from the Database.
- If the parent object has any another child then hibernate is going to throw an exception.
- If the parent object of the child object does not have no more childs then hibernate deletes the child object and also its parent object from the Database.
- If you want to delete only the child object by without effecting it on its parent object then we need to modify the cascade attribute to save (or) save-update, but not delete (or) all.

Eg:-

```
Customer c=(Customer) session.get(Customer.class, 501);
```

```
Transaction tx=session.beginTransaction();
```

```
session.delete(c);
```

```
tx.commit();
```

## One-to-many (Bi-directional) / many-to-one (bi-directional)

- One-to-many (bi-directional). is a combination of one-to-many and many-to-one.
- For one-to-many, we need a collection in parent class. and for many-to-one, we need a reference of parent is required in child class.
- for Eg:- If we want to apply one-to-many in bi-directional b/w vendor and Customer pojo classes then we need the following changes.
- ① Create a collection property in parent class.
- ② Create a reference of parent in child class
- ③ add ~~<set>~~ <collection type of tag> in parent mapping file and add <many-to-one> tag in child mapping file.

// Parent

```
public class Vendor
{
    private int vendorId;
    private String vendorName;
    private Set customers;
    .....
}
```

Vendor.hbm.xml

// child

```
public class Customer
{
    private int customerId;
    private String customerName;
    private String customerAddress;
    private Vendor vendor;
    .....
}
```

<h-m>

```
<class name="Vendor" table="Vendor">
    <id ... ... />
    <property ... ... />
    <set name="customers" cascade="all" inverse="true">
        <key column="venid"/>
        <one-to-many class="Customer"/>
    </set>
</class>
</h-m>
```

- In the above mapping file, for `<set>` tag, we added an attribute "inverse=true". This inverse attribute is used to inform the hibernate that the relationship is bi-directional.
- The default value of inverse attribute is "false".
- If we don't add "inverse=true" then hibernate uses more database operations to apply bi-directional relationship.

### Customer.hbm.xml

`<h-m>`

`<class name="Customer" table="customer">`

`<id -->`

`<property name="..."/>`

`<property name="..."/>`

`<many-to-one name="Vendor" class="Vendor" column="venid" cascade="all"/>`

`<class>`

`</h-m>`

for one-to-many (bi-directional) page no: ⑥ an application (6) of handout ②.

### many-to-many relationship :-

- many-to-many relationship is a one-to-many relationship from both sides.
- To apply many-to-many relationship b/w objects of two POJO classes using hibernate then we need a collection type of property at both POJO classes.
- many-to-many relationship is always a bi-directional. we no need to inform the hibernate that it is bi-directional.

Eg:-

We have two POJO classes called Student and Course. The many-to-many relationship between the two POJO classes is, one Student joined in multiple courses, i.e. one-to-many. In reverse, one course joined by multiple students, so again it is one-to-many. Finally the relationship is many-to-many.

- In Student POJO class, we need a collection type of property for storing a group of courses joined by a Student. Similarly, in Course POJO class, we need a collection type of property for storing a group of students joined in a Course.
- While applying many-to-many relationship b/w two POJO classes objects, at database side, we need a third table also, called as Join table.
- The minimum no. of tables required to apply many-to-many relationship is 3.
- A Join table contains foreign keys of many-to-many relationship.

Example :- The POJO classes and the Database tables of this student and course relationship are

public class Student  
{  
    → StudentId  
    → StudentName  
    → StudentGpa  
    private Set courses;  
}

public class Course  
{  
    → courseId  
    → CourseName  
    → duration  
    private Set students;  
}

PK  
↓  
Student  
Bid sname Grp

PK  
↓  
Course  
Cld cname duration

Students\_Courses (Join table)

Sid\_pk Cid\_pk

→ Student.hbm.xml

<h-m>

<class name="Student" table="student">

Lid

<property name="Lid" type="int" id="true" column="id" />

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

\*\*\* <set name="courses" cascade="all" table="Students\_Courses" />

<key column="Sid\_pk" />

<many-to-many class="Course" column="Cid\_pk" />

↳ opposite foreign key

</set>

</class>

<h-m>

→ Course.hbm.xml

<h-m>

<class name="Course" table="course">

Lid --> primary key of Course table, no inheritance, no association

<property name="Lid" type="int" id="true" column="id" />

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

<set name="Students" cascade="all" table="Students\_Courses" />

<key column="Cid\_pk" />

<many-to-many class="Student" column="Sid\_pk" />

</set>

</class>

<h-m>

→ In a client application of Hibernate, the following is the procedure for adding a group of courses to a student.

```
Set set1 = new HashSet();
set1.add(c1);
set1.add(c2);
s1.setCourses(set1);
```

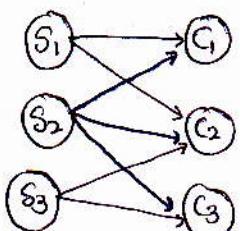
```
Set set2 = new HashSet();
set2.add(c1);
set2.add(c2);
set2.add(c3);
s2.setCourses(set2);
```

→ In the above code, two courses are added for student1 and three courses are added for student2. So, the relationship is one-to-many. At the same time, same course is added for student1 and student2. So, again the relationship is one-to-many. Finally the relationship added is many-to-many.

In a client appn of hibernate, the following is the procedure for adding a group of students to a one course.

```
Set set1 = new HashSet();
set1.add(s1);
set1.add(s2);
c1.setStudents(set1);

Set set2 = new HashSet();
set2.add(s1);
set2.add(s2);
set2.add(s3);
c2.setStudents(set2);
```



many-to-many is nothing but, one-to-many relationship  
again reverse of one-to-many relationship.

S<sub>1</sub> student joined in C<sub>1</sub> & C<sub>2</sub> courses → one-to-many  
C<sub>1</sub> course is joined by S<sub>1</sub> & S<sub>2</sub> student → one-to-many

### Many-to-many

→ Student.java

Course.java

Student.hbm.xml

Course.hbm.xml

hibernate.cfg.xml

InsertClient.java

\*.class

refer page ⑧ application no: 17

#### Output:-

SQL> Select \* from Student;

<u>SID</u>	<u>SNAME</u>	<u>GPP</u>
1001	aaa	Btech
1002	bbb	MCA

SQL> Select \* from course;

<u>CID</u>	<u>CNAME</u>	<u>DURATION</u>
902	oracle	100
901	java	150

SQL> Select \* from Students\_Courses;

<u>STD.FK</u>	<u>CID_FK</u>
1001	902
1001	901
1002	902
1002	901

#### ④ Adding an additional course to a Student :-

→ To add a new course to the existing Student then the following "4" steps are required.

Step-I:- load a Student Object from the DataBase.

Step-II:- read the set of Courses of the Student

Step-III:- Create a new Course Object with data

Step-IV:- add the Course Object to the set with in a Transaction.

||Step-1:-

Student s = (Student) session.get(Student.class, 1001);

||Step-2:-

Set set = s.getCourses();

||Step-3:-

Course c = new Course();

c.setCourseId(903);

c.setCourseName("CPP");

c.setDuration(150);

||Step-4:-

Transaction tx = session.beginTransaction();

set.add(c);

tx.commit();

SEL > select \* from Course;

CID	CNAME	Duration
902	oracle	100
901	java	150
903	CPP	150

SEL > select \* from students\_courses;

SID_FK	CID_FK
1001	901
1002	901
1001	902
1002	902
1001	903

To add a new Student to the existing Course then the following 4 steps are required.

Step-I:- load a Course object from the Database

Step-II:- read the set of students of the course

Step-III:- create a new student object with data

Step-IV:- add the student object with the set with in a Transaction.

Step-I:-

Course c = session.get(Course.class, 901);

Step-II:-

Set set = c.getStudents();

Step-III:-

Student s = new Student();

s.setStudentId(1003);

s.setStudentName("xyz");

s.setStudentGPA("B+");

Step-IV:- Transaction tx = session.beginTransaction();

set.add(s);

tx.commit();

## Deleting a Course from a Student :-

→ To delete an existing course from a student the following steps are required.

- ① load the student from DataBase
- ② read all the courses joined by a collection object
- ③ load the course to be removed from a student
- ④ remove that course from the collection, with in a Transaction.

|| Step-1

```
Student s = (Student) session.get(Student.class, 1001);
```

|| Step-2

```
Set set = s.getcourses();
```

|| Step-3

```
Course c = (Course) session.get(Course.class, 902);
```

|| Step-4

```
Transaction tx = session.beginTransaction();
set.remove(c);
tx.commit();
```

## \* One-to-one :-

one-to-one relationship can be applied in two ways.

- ① one-to-one with Foreign key
- ② one-to-one with Primary key

### One-to-one with Foreign key:-

- One-to-one with Foreign key is almost equal to applying many-to-one, but the difference is Foreign key column of child table should not allow duplicate values and also null values.
- While configuring this one-to-one relationship with Foreign key, we use <many-to-one> tag in the mapping file, but we add "Unique" & "not-null" attributes for the <many-to-one> tag.
- While applying this one-to-one with Fk relationship, we need a reference in child class pointing to its parent class.

### Example:-

We have two POJO classes called Person and License. The relationship is,

one person has one License.

- In hibernate, while applying this one-to-one relationship, the relationship must be configured from child class to parent class.

→ The POJO classes of this one-to-one relationship are

public class Person

{

private int personId;

private String personName;

setters & getters

}

PK  
↓  
person  
pid pname

public class License  
{  
private int licenseId;  
private String issuedDate;  
private Date expiredDate;  
private Person person;  
getters & setters  
}

PK  
↓  
license  
lid idate edate period  
FK  
↓

// person.hbm.xml

<h-m>

<class name="Person" table="person">

<id -->

<property -->

<property -->

</class>

</h-m>

// license.hbm.xml

<h-m>

<class name="License" table="license">

<id -->

<property -->

<property -->

(\*\*\*) <many-to-one name="person"

↓  
reference  
name

class="Person"

↓  
parent  
class

column="perId"

L\_Fk

(\*\*) unique="true"

(\*\*) not-null="true"

cascade="all" />

</class>

</h-m>

### OneToOneFk

↳ person.java  
↳ license.java  
↳ person.hbm.xml  
↳ license.hbm.xml  
↳ hibernate.cfg.xml  
↳ InsertClient.java  
↳ \*.class

O/P:- Select \* from person;

pid	pname
1101	ABC

Select \* from license;

Lid	ldate	Exdate	perId
1010	28-mar-13	28-mar-13	1101

refer page (ii) on appn 18 of handout 2.

→ In this no additional columns are added bcz foreign key column does not allow duplicate values.

→ In this delete operation is performed when the parent object is removed from the table automatically child obj will be removed from the table.

### One-to-one with Primary key :-

- In one-to-one with Foreign key, in child table a separate foreign key column is taken (required) and it does not allow duplicate & null values
- In child table, there is a primary key column also, which does not allow duplicate and null values.
- Instead of taking primary key and Foreign key separately, we can make primary key column also as a Foreign key in the child table. This type of relationship is called one-to-one with primary key.
- In this one-to-one relationship, primary key value of parent object will be copied into primary key of child object.
- In this one-to-one with primary key, we use a generator class called "foreign"
- With the help of "foreign" generator, we can inform the hibernate that copy the primary key value of a parent object into primary key value of the child object.

```
public class Person
{
    private int personId;
    private int personName;
    setters & getters
}
```

// person.hbm.xml

<h-m>

<class name="Person" table="person">

<id -->

<property -->

</class>

<h-m>

```
public class License
{
    private int licenseId;
    private Date issuedDate;
    private Date expiredDate;
    private Person person;
    setters & getters
}
```

/ License.hbm.xml

<h-m>

<class name="License" table="license">

<id name="licenseId" column="lid">

<generator class="foreign">

<param name="property">person</param>

</generator>

<id>

<property -->

<property -->

<one-to-one name="person" class="Person" cascade="all"/>

<h-m>

Pk  
 ↓  
Pid  
person (parent)

PK&FK  
 ↓  
lid    idate    edate  
 license (child)

One to One Pk

→ person.java

license.java

license.hbm.xml

person.hbm.xml

hibernate.cfg.xml

InsertClient.java

\*.class

refer page 13 an appn 19 of handout 2

program is not inserting into jointable.  
hibernate inserts into jointable.

### <idbag> of many-to-many :-

- While applying many-to-many relationship, hibernate uses Join table for storing foreign keys. In the join table the foreign keys values are duplicated.
- In a join table, there is no uniqueness (or) uniquecolumn, to access (or) modify the data easily. So, for adding uniqueness ~~pk~~ to the join table, we use `<idbag> tag`.

- To configure `<idbag>` tag in a mapping file, the collection type in the pojo class must be "List" (`java.util.List`)
- in `<idbag>` tag, we use another tag called `<collection-id>` for mapping the primary key column of join table.

The following are the changes required in previous many-to-many example we use `<idbag> tag`.

- ① In `Student.java`, replace ~~Set~~ collection type Set with collection type List.

```

import java.util.*;
public class Student
{
  private int studentId;
  private String studentName;
  private String studentGnp;
  private List courses;
}
  
```

② in Student.hbm.xml, replace <Set> tag with <idbag> tag like the following.

```
<idbag name="courses" table="Students-Courses" cascade="all">
    <collection-id column="S_C_ID" type="int" />
    <generator class="increment" />
    </collection-id>
    <key column="Sid_pk" />
    <many-to-many class="Course" column="Cid_pk" />
</idbag>
```

NOTE:- In the above <Collection-id> tag, the generator class should not be "assigned".

"assigned".

→ though we are using Set object, we need list object.

③ in InsertClient.java, in place of <Set> object we need list object.

```
List list = new ArrayList();
```

```
list.add(c1);
```

```
list.add(c2);
```

```
S1.setCourses(list);
```

```
S2.setCourses(list);
```

④ After executing the InsertClient, we obtain the following output in DataBase.

Select \* from Student;

Sid	sname	grp
1001	aaa	Btech
1002	bbb	MCA

Select \* from course;

Cid	cname	Duration
901	java	180
902	oracle	100

Select \* from Students-Courses;

Sid_pk	Cid_pk	S_C_ID
1001	901	1
1001	902	2
1002	901	3
1002	902	4

\*\* Q:- What is  $1+n$  (or)  $n+1$  problem in hibernate?  
Ans:- In relationships, when loading ~~a~~ parent objects with their respective child objects from the Database then hibernate uses 1 select operation for loading all the parent objects and hibernate uses a separate select operation for loading child objects of each parent object. It means, if there are 5 parent objects then hibernate uses "1" select operation for loading all the 5 parent objects and 5 select operations for loading the childs of each parent object. Totally  $1+5$  select operations are generated by hibernate. This is called " $1+n$ " problem in hibernate.

NOTE:- In  $1+n$  problem ~~as~~ "n" represents no. of parent objects.

### Joins :-

- Join statements are used to select the data of multiple POJO class objects, by using the relationship added b/w them, using a single select operation.
- Join statements are only applicable, when a relationship exist between the POJO class objects.
- Hibernate Supports ~~the~~ following 4 types of Joins.
  - ① Inner Join
  - ② Left ~~Outer~~ Join
  - ③ Right Outer Join
  - ④ Full Join
- The following HQL query is to Select a Vendor name and its associated Customer name using One-to-many relationship added b/w a Vendor and a Customer objects.

```
Select v.vendorName, c.customerName from Vendor v join v.customers c  
      ↓ collection property
```
- The following HQL Command is for Select a Vendor name and its associated Customer name, by using many-to-one relationship b/w

## Customer and Vendor.

Select v.vendorName, c.customerName from Customer c join c.vendor

→ If we don't specify a join type then by default the join type is "InnerJoin".

→ Inner Join is an equiJoin of SQL. It means, the join stmt selects only the data from table, if an associated data exist for it.

\* The following client appn uses one-to-many relationship b/w a vendor and a customer pojo classes. and obtains the data from two tables using join stmt

// JoinTest.java

```
import java.util.*;  
import org.hibernate.*;  
import org.hibernate.cfg.*;
```

```
public class JoinTest
```

```
{
```

```
    public void m (String[] args)
```

```
{
```

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

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

Query qry = session.createQuery ("select v.vendorName, c.customerName from Vendor  
join v.customers c");

```
    List l = qry.list();
```

```
    Iterator it = l.iterator();
```

```
    while (it.hasNext())
```

```
{
```

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

```
        System.out.println (row[0] + " " + row[1]);
```

```
}
```

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

```
    factory.close();
```

```
}
```

```
}
```

→ In case of LeftOuter join, Hibernate selects equal data from both sides of the join and also more data from Left side of the join, even though there is

no associated data at Right side of the Join.

each side of parent each

for Eg:-, The following Join statement selects more data from Vendor, even though there is no associated data at Rightside (i.e. <sup>at</sup>Customer)

Select v.vendorName, c.customerName from Vendor v left outer join v.customers c

- In Case of right Outer join, hibernate selects more data from right side of the join, even though there is no associated data at Left Side of the join.
- In case of Full Join, hibernate selects both equal data and also unequal data from both sides of the Join statement.

Solution for 1+n problem :- in HQL

- To reduce 1+n select operations to a single Select operation, for loading the parent objects and their associated child objects, we need to use "join fetch" statement in HQL.
- If we construct the following HQL query then hibernate selects all vendors and their respective customers with a single Select operation.  
Query qry = session.createQuery ("from Vendor v join fetch v.customers");  
L, it is automatically disable the Lazy loading.
- If we execute the above query then hibernate automatically disables lazy loading (nothing but early loading) and generates one select operation for loading all parents and childs  
By reducing the no.of Select operations, the performance of an application will be increase.

Solution for 1+n problem in Criteria :-

- The following is for reducing 1+n problems through Criteria API of hibernate.  
Criteria crit = session.createCriteria(Vendor.class);  
crit.setFetchMode("customers", FetchMode.JOIN);  
List l = crit.list();

In the above code, if we don't set Fetch mode then by default the Fetch mode will be taken as "SELECT". SELECT mode means "1+n" Select Operations are generated to reduce "1+n" Select Operations into a Single Select operation, we need to change the fetch mode as "JOIN".

### Fetching Strategies of Hibernate :-

- ① Select
- ② Join
- ③ Subselect

- ① → by default fetching strategy is "select". select means, one select operation is generated for reading the parents and a separate select operation is generated for reading childs of each parent. here, 1+n problem occurs.
- ② → "Join" strategy generates "1" select operation for all parents and their respective collection of childs, to load from the Database. it means "1+n" operations are reduced to "1" select operation.
- ③ → "subselect" fetching strategy, hibernate generates "1" select operation for all parent objects and "1" select operation for all childs of all parents. totally "1+1" select operations are generated.

- \*) If we call SetFetchMode (-,-) method of Criteria then it is possible to set either "Select (or) Join" fetching strategy. But subselect fetching strategy can't be set.
- \*) In stead of calling SetFetchMode (-,-) method, we can configure the fetching strategy in a mapping file, by adding "fetch" attribute for the relationship configuration of mapping file.

```
<set name="customers" cascade="all" fetch="select|join|subselect">  
  ...  
</set>
```

## NetBeans IDE :-

Steps to develop an Hibernate appn using NetBeans IDE.

Step-1:- Create a table in the Database like the following.

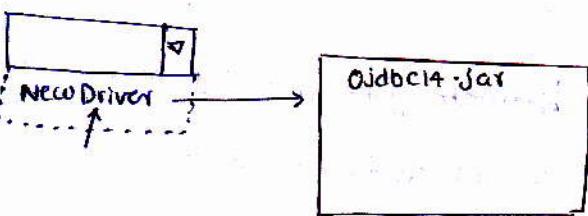
SQL> create table demo (sid number(5) primary key, sname varchar2(10));

Step-2:- Start NetBeans IDE

Step-3:- Click on File Menu → New Project → Java → Java Application → next  
→ project Name: **Hibernate1** →  create Main class → Finish

Step-4:- Right click on Libraries folder → Add Library → Hibernate → Add Library

Step-5:- Right click on project name (Hibernate1) → new → Other → Hibernate →  
Hibernate Configuration Wizard → next → next → Database Connection:  
→ New Database Connection → enter the following details →

Driver Name :  → add → OK

Host : **localhost**  
port : **1521**  
ServiceID: **orcl**  
Username: **scott**  
password: **tiger**

Step-6:- Right click on project name → new → Other → Hibernate →  
Hibernate Reverse Engineering Wizard → next → next → Select Demo table  
at left side → add → finish

Step-7:- Right click on project name → new → Other → Hibernate → select  
Hibernate Mapping Files and POJOs from Database → package: **P1** → finish

Step-8:- Right Click on project name → new → Java Class → class Name: **Insertclient**  
→ finish

```
import org.hibernate.*;  
import org.hibernate.cfg.*;
```

```

public class InsertClient {
    public void main (String args[]) {
        SessionFactory factory = new Configuration().configure().BuildSessionFactory();
        Session session = factory.openSession();
        P1.Demo d = new P1.Demo();
        d.setId(111);
        d.setName("abc");
        Transaction tx = session.beginTransaction();
        session.save(d);
        tx.commit();
        session.close();
        factory.close();
    }
}

```

Step-9:- Right click on InsertClient.java → RunFile

### Annotation :-

① // This is used in String format → metadata  
 String s1 = "10" → data

② <sup>metadata</sup> Final class Demo  
 {  
 int a=10; } data  
 String s = "100"; }  
 }

③ hibernate.cfg.xml  
<property name="org.hibernate.Connection"> Oracle.jdbc.OracleDriver </property>  
 ↓ data

④ hbm.xml  
<class> ... </class>  
<id> ... </id>  
<property> ... </property>

it contains  
totally metadata

To reduce the mapping file burden on XML file we use Annotation. bcz it contains only metadata. Only metadata content will be write in Annotation format. Data / metadata content does not write in Annotation format. so, hibernate mapping file mostly write in Annotation format, but not configuration file write in Annotation format.

## Annotations in Hibernate

- While writing Java apps, there are different ways to provide metadata.
- ① We can write comments in a Java program, define some kind of metadata for Java statements in a program.
- ② In Java, we have access modifiers, which also defines metadata for the elements of a Java application.
- ③ We can create XML file to define metadata.
- In the above 3 ways, XML way of defining metadata is increasing the burden on Java programmers, Because as part of a Java Project development, a Java application need more no. of XML files.
- In order to reduce the heavy use of XML's in a Java Project development, we got annotations, introduced into java from Jdk1.5
- While constructing Hibernate applications, we are providing ORM metadata in the form of mapping XML files.
- In order to reduce the mapping XML files from an Hibernate apps, we got annotations in Hibernate.
- By using annotations, we can't replace because a Configuration file is going to contain data also.
- Annotations in Hibernate for persistence are the implementations of the annotations provided by SUN as part of Java Persistence API (JPA).
- JPA is a specification released by SUN as part of EJB3.x and implementations of this specification are provided by the Server vendor supporting EJB3.x and ORM tool vendors.
- The annotations in all ORM tools for persistence are same in all ORM tools, because the all ORM tools are provided implementation for common JPA Specification.

→ While creating an annotation, some metadata is provided for an annotation.

It means metadata of metadata.

The following are the two informations added for an annotation while creating it.

① To what type of Java element an annotation is applicable. That is for a class (or) for a method (or) for a property... etc.

② To what level this annotation is visible. i.e. whether an annotation is visible for Compiletime level (or) Runtime level.

→ The 1<sup>st</sup> information for an annotation is set by using `@Target`. This is called Target Specification (or) Target Information.

→ The 2<sup>nd</sup> information is added using `@Retention`. This is called setting the Retention policy.

Common Annotations with POJO class :-

→ While creating a pojo class of hibernate, we need to import a package of JPA called `javax.persistence`.

→ The following 4 annotations are commonly used while creating a pojo class.

① `@Entity`

② `@Table`

③ `@Id`

④ `@Column`

`@Entity` and `@Table` are applicable at class level and `@Id` & `@Column`

are applicable at either property level or setter method level.

\*\* For each pojo class, we should add `@Entity` annotation. By adding this annotation, a pojo class acts like an Entity bean.

→ `@Table` is used to map a pojo class to a database table. If pojo class name and table name are matched then this `@Table` annotation is optional.

@Id is mapped to a property with a primary key column of a DataBase table.

→ @ Column is to map a property with a Column of a DataBase table.

→ If property name and column name are matched then @ Column annotation is optional.

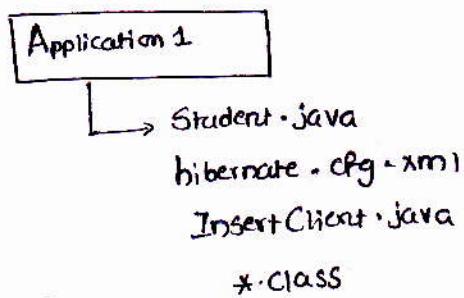
→ In configuration file of hibernate, instead of mapping resource, we should add mapping "class".

→ In case of before hibernate 3.6, in a client appn we need AnnotationConfiguration class object. but from hibernate 3.6 AnnotationConfiguration class is merged into Configuration class only. so, we don't required any changes in a client application of hibernate.

NOTE:-  
(i) In case of annotations, annotations are added to POJO class. so, If any changes are made ~~to~~ to Annotation then we need to recompile POJO class. but this is not a drawback. The reason is, in an application metadata is not changed frequently. only data changes frequently.

(ii) If we use annotations and also we create mapping XML file for metadata then finally mapping XML metadata is considered.

① The following appn is for inserting a Student Object in a DataBase using Annotations.



refer page ⑯ an appn ⑰ of handout ⑲

Q:- Difference b/w save(-) & persist(-) method?

Ans:- both save(-) & persist(-) methods are used to save an object in a DataBase. but save(-) method returns the id of the saved object back to a Java program, but persist(-) method doesn't return the id of saved object.

→ return type of save(-) method is Serializable, but the return type of persist(-) method is Void.

→ Save(-) method is useful, to see the generatedId for an object, when generator class is used as other than "assigned".

→ But while calling save(-) method, we can store the returned value into a Serialized object.

Integer i = (Integer) session.save(s);

↳ serialized object

→ If we use generator is assigned, then persist(-) method is written. It doesn't return any object, bcz its return type is Void.

## Annotations for inheritance mapping :-

- for inheritance mapping, we need a class level annotation called @Inheritance.
- For @Inheritance, we need to pass a parameter called "strategy". This strategy param value indicates the inheritance type.

@Inheritance (strategy = InheritanceType. SINGLE\_TABLE) → Table per class

@Inheritance (strategy = InheritanceType. JOINED) → Table per subclass

@Inheritance (strategy = InheritanceType. TABLE\_PER\_CLASS) → Table per concrete class

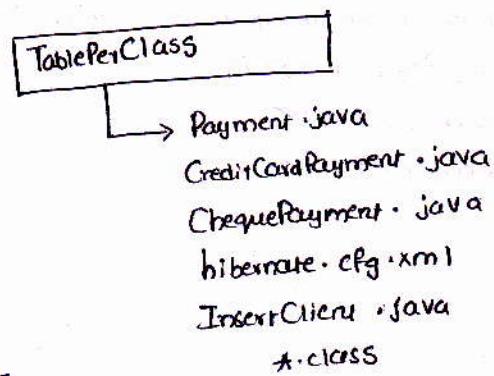
### Table per class :-

- When Table per class hierarchy is applied then we need a Discriminator Column in the DB table

- To map a Discriminator Column, we use

@DiscriminatorColumn and to assign the value, we use @DiscriminatorValue.

- In case of Table per subclass (or) Table per concrete class, Discriminator Column is optional.



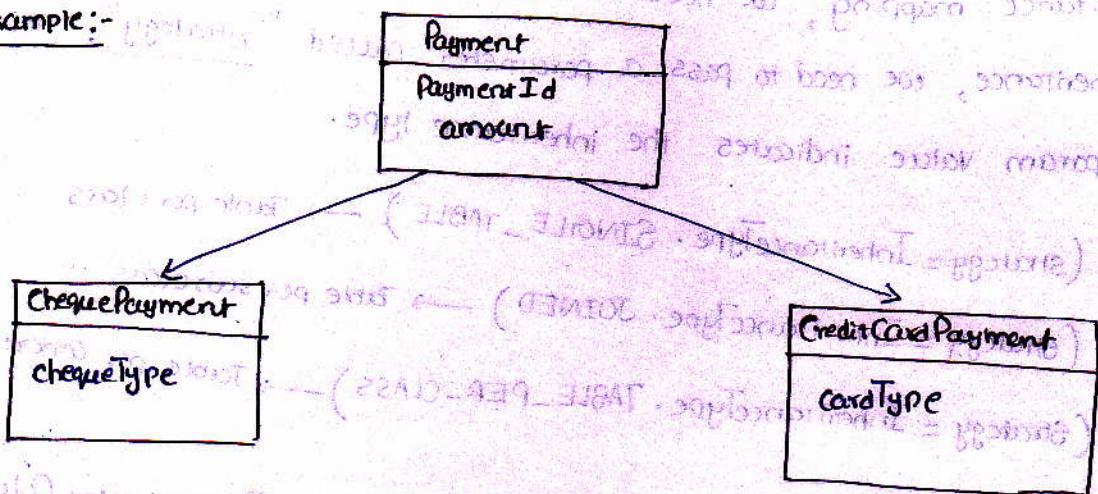
refer page no: 17 an application  
② of an Handout ②

### Table Per Subclass :-

- In case of Table Per Subclass hierarchy, in the DataBase separate tables are required for parent class and also for each sub class.
- At the time of inserting the objects, Hibernate copies Parent properties into Parent table and child properties into child table and copies primary key of parent table into child table.
- In child table, the primary key of the child table acts like a Foreign key also.

→ To inform the hibernate that a column is acting like a primary key and also a Foreign key, we need an annotation called "@PrimaryKeyJoinColumn".

Example:-



PK → payment → payid

Credit-table  
PK → Pid  
cheque-table  
PK → pid

TablePerSubclass

→ Payment.java

CreditCardPayment.java

ChequePayment.java

InsertClient.java

hibernate.cfg.xml

## Annotations With Relationships

### One-to-many with Annotations :-

→ The following two Annotations are required for applying one-to-many (unique) relationship

① `@OneToMany (targetEntity = class obj of child pojo class, cascade = CascadeType.ALL, fetch = FetchType.EAGER)`

② `@JoinColumn (name = "fk", referencedColumnName = "pk") LAZY`

#### OneToMany Annotation

→ Vendor.java

Customer.java

hibernate.cfg.xml

InsertClient.java

\*.class

refer page no (20) on appn (24)  
of handout (2)

### many-to-one with Annotations :-

→ The following two Annotations are required for applying many-to-one (unique) relationship.

we need a property in child class, which is pointing to the its parent class.

→ To the parent class reference property, we need to apply the following two Annotations.

① `@ManyToOne (targetEntity = class obj of parent pojo class, cascade = CascadeType.ALL, fetch = FetchType.EAGER)`

② `@JoinColumn (name = "fk", referencedColumnName = "pk")`

for Example, to apply many-to-one relationship from Customer to Vendor, we need to create the Vendor and Customer POJO classes like the following.

```
//Vendor.java
import javax.persistence.*;
@Entity
@Table(name = "vendor") => optional
public class Vendor
{
    @Id
    @Column(name = "vid")
    private int vendorId;
    @Column(name = "vname");
    private String vendorName;
}
```

→ Scans & get us

```
//Customer.java
import javax.persistence.*;
@Entity
@Table(name = "customer")
public class Customer
{
    @Id
    @Column(name = "custId")
    private int customerId;
    @Column(name = "curname")
    private String customerName;
    @Column(name = "custAdd")
    private String customerAddress;
```

In XML format we have proxy loading feature loading.

But in Annotation we don't have proxy loading.  
We have LAZY (or) EAGER Loadings

@ManyToOne (targetEntity = Vendor.class, cascade = CascadeType.ALL);  
@JoinColumn (name = "vid", referencedColumnName = "vid");

private Vendor vendor;

}

### ManyToOne Annotation

↳ Vendor.java

Customer.java

hibernate.cfg.xml

InsertClient.java

\*.class

One-to-many (bi-directional) with Annotation :-

- By adding both one-to-many and many-to-one annotation, we can develop bi-directional one-to-many association.
- While creating bi-directional relationship, instead of repeating @JoinColumn annotation for both sides, we can avoid this @JoinColumn at many-to-one by adding "mappedBy" parameter for @ManyToOne(-,-,-,-) annotation.
- To apply one-to-many in bi-directional b/w Vendor and Customer we create Vendor and customer classes like the following.

```
public class Vendor {  
      
    @Id  
    @Column(name = "vid");
```

private int VendorId;

@Column (name = "vname");

private String vendorName;

@OneToMany (targetEntity = Customer.class, cascade = CascadeType.ALL)

@JoinColumn (name = "vid", referencedColumnName = "vid")

private Set Customers;

}

```

public class Customer
{
    → customer Id
    → customer Name
    → customer Address

```

@ManyToOne (targetEntity = Vendor.class, cascade = CascadeType.ALL,  
mappedBy = "customers")

```
private Vendor vendor;
```

```

    =
    {
        ...
    }
}
```

→ To Avoid @JoinColumn in child table.

### Many - TO - Many with Annotations :-

- Many - TO - Many relationship is always a bi-directional.
- When many - TO - Many relationship is applied then we need a JoinTable also in the DataBase.
- A minimum of 3 tables are required to apply the many - to - many relationship.
- The following are the 3 Annotations required for applying Many - to - Many Association.
  - 1) @ManyToMany
  - 2) @JoinTable
  - 3) @JoinColumn
- In case of ManyToMany, we need JoinTable and the foreign keys of the relationship are stored in JoinTable.
- @JoinColumn annotation is used inside @JoinTable annotation.
- Eg:- We have two POJO classes called Category & Item. The relationship b/w these two class objects is Many - to - Many.  
i.e. one Category has Many items and one item belongs to Many Categories.
- The DataBase tables of this relationship are

Categories	
cat_id	cat_name
categories - items	
cat_id_pk	item_id_pk

items	
item_id	item_name
categories - items	
item_id_pk	cat_id_pk

// Category.java

```
import javax.persistence.*;
import java.util.*;
```

@ Entity

@ Table (name = "categories")

```
public class Category
```

{

@ Id

@ Column (name = "cat\_id")

```
private int category_id;
```

@ Column (name = "cat\_name")

```
private String categoryName;
```

@ ManyToMany (targetEntity = Item.class, cascade = CascadeType.ALL)

@ JoinTable (name = "categories\_items", jointable = true table

joinColumns = @ JoinColumn (name = "cat\_id\_pk", referencedColumnName = "cat\_id")

inverseJoinColumns = @ JoinColumn (name = "item\_id\_pk", referencedColumnName = "item\_id")

```
private Set items;
```

→ Setters & getters

}

// Item.java

```
import javax.persistence.*;
import java.util.*;
```

@ Entity

@ Table (name = "items")

```
public class Item
```

{

@ Id

@ Column (name = "item\_id")

```
private int item_id;
```

@ Column (name = "item\_name")

```
private String itemName;
```

@ ManyToMany (targetEntity = Category.class, cascade = CascadeType.ALL, mappedBy = "items")

```
private Set categories;
```

g

≡

no need to write @JoinTable once again

-1

↓  
Property of category  
(Collection)

## ManyToMany Annotation

→ Category.java

Item.java

hibernate.cfg.xml

InsertClient.java

refer page no: 24 an appn 26

## \* One-to-one Using Annotations:-

→ We can apply one-to-one relationship in two ways.

① one-to-one with Foreignkey

② one-to-one with Primarykey

To apply one-to-one with Foreignkey we should not allow duplicate values and null values into Foreignkey column.

While applying one-to-one using Foreign key with annotation, we should add Unique and notnull attributes for @JoinColumn annotation.

For one-to-one with Foreignkey, we need @ManyToOne annotation only.

If you want to apply one-to-one relationship b/w person and License classes using annotations, in one-to-one with Foreignkey approach then the following are the classes with annotations.

```
// Person.java  
import javax.persistence.*;  
  
@Entity  
@Table(name = "person")  
public class Person  
{  
    @Id  
    @Column(name = "pid")  
    private int personId;  
  
    @Column(name = "pname")  
    private String personName;  
    ...  
}
```

```
/* License.java */  
import javax.persistence.*;  
  
@Entity  
@Table(name = "license")  
public class License  
{  
    @Id  
    @Column(name = "lid")  
    private int licenseId;  
  
    @Column(name = "idate")  
    private Date issuedDate;  
  
    @Column(name = "edate")  
    private Date expireDate;  
  
    @ManyToOne(targetEntity = Person.class, cascade = CascadeType.ALL)  
    @JoinColumn(name = "per_id", referencedColumnName = "pid",  
               unique = "true", nullable = "false")  
    private Person person;  
    ...  
}
```

## adding a generator for an id using annotations:-

- If we want to add any ~~new~~ hibernate provided generator class to an Id then first we need to define ~~that~~ generator Strategy using `@GenericGenerator`
- after defining the strategy , to add a generator for an Id we need `@GeneratedValue` annotation.
- `@GenericGenerator` is an ~~both~~ <sup>own</sup> annotation of hibernate and it is not from JPA.
- Hibernate has provided two annotations of its own `@GenericGenerator` & `@Parameters`.
- These two annotations are from org.hibernate.annotations pkg.
- If you don't specify any generator for an Id by default assigned generator is used.

- If we want to add "Increment" generator for an Id using annotations then we should create POJO class like the following

```
// product.java
import javax.persistence.*;
import org.hibernate.annotations.*;

@Entity
@Table(name = "product")
public class Product
{
    @GenericGenerator(name = "gen1", strategy = "increment")
    @Id
    @GeneratedValue(generator = "gen1")
    @Column(name = "pid")
    private int productId;
    ...
}
```

- To apply "Sequence" generator for an Id , we need the following annotations
  - `@GenericGenerator ( name = "gen1", strategy = "sequence", parameters = { @Parameter ( name = "sequence", value = "my sequence" ) } )`
  - `@Id`
  - `@GeneratedValue(generator = "gen1")`
  - `@Column(name = "pid")`
  - `private int productId;`

→ To apply "hilo" generator for an Id, we need the following generators.

@GenericGenerator(name = "gen1", strategy = "hilo",

parameters = {@Parameter(name = "table", value = "myTable"),  
@Parameter(name = "column", value = "next"),  
@Parameter(name = "max\_lo", value = "10")})

@Id

@GeneratedValue(generator = "gen1")

@Column(name = "pid")

private int productId;

...

### OneToOne with Primary key :-

→ To apply OneToOne with primary key, we need to make primary key of child table also as a foreign key.

→ To inform the hibernate that, copy the primary key of one object into primary key of another object, we use "foreign" generator.

→ for example, To apply OneToOne with primary key relationship b/w a person and a license class, we need to create the POJO classes with annotations like the following.

//Person.java

import javax.persistence.\*;

@Entity

@Table(name = "person")

public class Person

{

@Id

@Column(name = "per\_id")

private int personId;

@Column(name = "per\_name")

private String personName;

=

}

//License.java

import java.util.\*; → Date class

import javax.persistence.\*; → JPA annotations

import org.hibernate.annotations.\*;

@Entity

@Table(name = "license")

public class License

{

@GenericGenerator(name = "gen1", strategy = "foreign")

parameters = {@Parameter(name = "property",  
value = "person")})

@Id

@GeneratedValue(generator = "gen1")

@Column(name = "lic\_id")

private int licenseId;

@Column(name = "idate")

private Date issueDate;

@Column(name = "edate")

private Date expireDate;

`@OneToOne(targetEntity = Person.class, cascade = CascadeType.ALL)`

`@PrimaryKeyJoinColumn` → it informs the hibernate that a primary key is private Person person; also acting like as a foreign key

### OneToOne Annotation

↳ Person.java  
License.java  
hibernate.cfg.xml  
InsertClient.java  
\*.class

refer page 26 appn 27 of handout 2

SQL → `Select * from person;`

PERIOD  
That

### Component mapping in hibernate :-

- Component mapping is a mapping of making a reference of one pojo class as a member variable of another pojo class.
- If you apply component mapping then Objects of two pojo classes are inserted to a Single table of DataBase.
- In order to inform the hibernate that one class reference is acting as a component of another class, In a hibernate mapping file we use `<Component>` tag
- If you want to apply component mapping using annotations then we use the following two annotations.

`@Embeddable`  
`@Embedded`

→ while using annotations, to make a pojo class as (`Embeddable`) insertable into another class object, we use an annotation `@Embeddable`.

→ In a POJO class, by making another class reference as a component, we use `@Embedded`

→ In the following example, a reference of PName class is acting as a component of Person class.

### //PName.java

```

import javax.persistence.*;
@Embeddable
public class PName
{
    @Column(name = "I")
    private char initial;
    @Column(name = "fname")
    private String firstName;
    @Column(name = "lname")
    private String lastName;
    ...
}

```

### //Person.java

```

import javax.persistence.*;
@Table(name = "persons")
public class Person
{
    @Id
    @Column(name = "PId")
    private int personId;
    @Column(name = "dob")
    private java.util.Date dob;
    @Embedded
    private PName pName;
}

```

### ComponentTest

#### Person.java

PName.java

hibernate.cfg.xml refer page (28) appn (28) of handout (2)

InsertClient.java

\*.class

SQL> select \* from persons;

PID	DOB	FNAME	I	LNAME
ABC	XYZ			
11	06-APR-13	Abc	S	

Q:- What are different types of OR mappings?

- Ans:-
- (1) Full Object mapping
  - (2) Partial Object mapping
  - (3) Inheritance mapping
  - (4) Component mapping
  - (5) Association / Relational mapping.

Q:- what is @Temporal annotation

Ans:- @Temporal annotation is used to inform the hibernate about whether a Date (or) Time (or) Date and Time are to be stored in the DataBase.

→ by default hibernate only inserts Date. If you want to inform Time (or) Time & Date is also inserted into DB then we apply this @Temporal annotation.

For eg:-

```

@Column(name = "dob")
@Temporal(TemporalType.TIMESTAMP)
private Date dob;

```

→ hibernate inserts only Time into DataBase from dob property.

for e.g:- @Column(name = "dob")

@Temporal(TemporalType.TIMESTAMP)

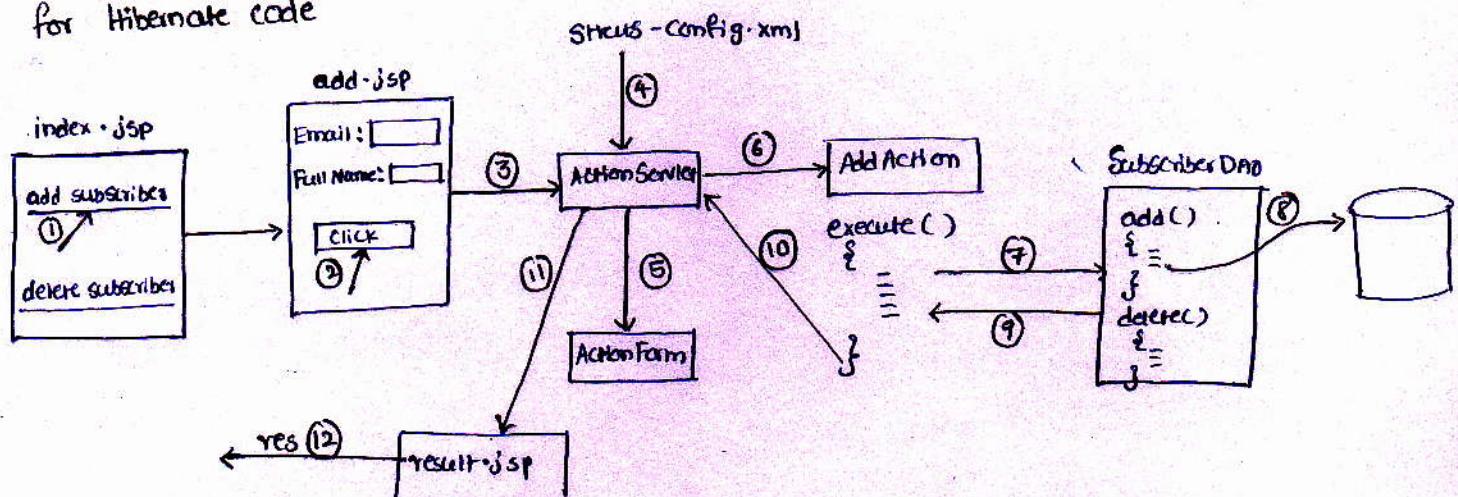
private Date dob;

In the above example, hibernate inserts both date and time into a DataBase table.

### Struts with Hibernate Integration :-

- Integrating Struts with Hibernate is nothing but calling persistence logic implemented in Hibernate from Action class of Struts.
- In Struts P/W, for integration is possible with other frameworks from Action class of struts.
- The following are the options available to integrate Struts with Hibernate.
  - ① we can define all the persistence logic of hibernate in execute(-,-,-) of Action class. In this approach, no.of hibernate Factory objects are increased. because execute(-,-,-) of Action class is called for each request.  
→ The drawback of this approach is, a burden on a server is increased, because of no.of SessionFactory objects.
  - ② we can create SessionFactory in Constructor of an Action class and Session in execute(-,-,-) of an Action class. Hence, factory is created for once, because in struts, ~~only~~ one object of Action class will be shared by multiple threads.  
→ The drawback of this approach is, Factory created in on Action class can't be shared with another Action class. It means, for example if we have 10 Action classes then we need to create 10Factories, one for each class.
  - ③ we can create a Separate java class for creating SessionFactory for once and for defining methods to do database operations by hibernate. This Separate class is called a DAO class.  
→ This approach is a better approach, for integrating Struts and hibernate. because One hibernate factory can be shared by multiple Actionclasses of Struts

\* The following is a Struts with Hibernate Integration example, where we are performing add and delete operations on a Subscriber of data, by ~~extending~~ <sup>Creating</sup> DAO class for Hibernate code



Subscribers

```
1 Application-1(Servlet with Hibernate: performing insert,update,delete and select  
2 operations (CURD operations))  
3 -----index.html-----  
4 <center>  
5 <h2>  
6 <a href="insert.html"> Create Employee </a> <br>  
7 <a href="update.html">Update Employee </a> <br>  
8 <a href="delete.html"> Delete Employee </a> <br>  
9 <a href="select.html"> Select Employee </a> <br>  
10 </h2>  
11 </center>  
12 -----insert.html-----  
13 <center>  
14 <h2>  
15   <form action="insertsrv">  
16     Empno : <input type="text" name="empno"><br>  
17     Ename : <input type="text" name="ename"><br>  
18     Sal : <input type="text" name="sal"><br>  
19     Deptno : <input type="text" name="deptno"><br>  
20     <input type="submit" value="INSERT">  
21   </form>  
22 </h2>  
23 </center>  
24 -----update.html-----  
25 <center>  
26 <h2>  
27   <form action="updatesrv">  
28     Empno : <input type="text" name="empno"><br>  
29     Ename : <input type="text" name="ename"><br>  
30     Sal : <input type="text" name="sal"><br>  
31     Deptno : <input type="text" name="deptno"><br>  
32     <input type="submit" value="UPDATE">  
33   </form>  
34 </h2>  
35 </center>  
36 -----select.html-----  
37 <center>  
38 <h2>  
39   <form action="selectsrv">  
40     Empno : <input type="text" name="empno"><br>  
41     <input type="submit" value="SELECT">  
42   </form>  
43 </h2>  
44 </center>  
45 -----delete.html-----  
46 <center>  
47 <h2>  
48   <form action="deletesrv">  
49     Empno : <input type="text" name="empno"><br>  
50     <input type="submit" value="DELETE">  
51   </form>  
52 </h2>  
53 </center>  
54 -----web.xml-----  
55 <web-app>  
56   <servlet>  
57     <servlet-name>s1</servlet-name>  
58     <servlet-class>InsertServlet</servlet-class>  
59   </servlet>  
60   <servlet-mapping>  
61     <servlet-name>s1</servlet-name>  
62     <url-pattern>/insertsrv</url-pattern>  
63   </servlet-mapping>  
64   <servlet>  
65     <servlet-name>s2</servlet-name>  
66     <servlet-class>UpdateServlet</servlet-class>  
67   </servlet>  
68   <servlet-mapping>  
69     <servlet-name>s2</servlet-name>  
70     <url-pattern>/updatesrv</url-pattern>  
71   </servlet-mapping>
```



```
72 <servlet>
73   <servlet-name>s3</servlet-name>
74   <servlet-class>DeleteServlet</servlet-class>
75 </servlet>
76 <servlet-mapping>
77   <servlet-name>s3</servlet-name>
78   <url-pattern>/deletesrv</url-pattern>
79 </servlet-mapping>
80 <servlet>
81   <servlet-name>s4</servlet-name>
82   <servlet-class>SelectServlet</servlet-class>
83 </servlet>
84 <servlet-mapping>
85   <servlet-name>s4</servlet-name>
86   <url-pattern>/selectsrv</url-pattern>
87 </servlet-mapping>
88 </web-app>
89 -----Employee.java-----
90 //Employee.java (POJO)
91 public class Employee
92 {
93     private int employeeId;
94     private String employeeName;
95     private int employeeSal;
96     private int deptNumber;
97     public int getDeptNumber() {
98         return deptNumber;
99     }
100    public void setDeptNumber(int deptNumber) {
101        this.deptNumber = deptNumber;
102    }
103    public String getEmployeeName() {
104        return employeeName;
105    }
106    public void setEmployeeName(String employeeName) {
107        this.employeeName = employeeName;
108    }
109    public int getEmployeeId() {
110        return employeeId;
111    }
112    public void setEmployeeId(int employeeId) {
113        this.employeeId = employeeId;
114    }
115    public int getEmployeeSal() {
116        return employeeSal;
117    }
118    public void setEmployeeSal(int employeeSal) {
119        this.employeeSal = employeeSal;
120    }
121 }
122 -----employee.hbm.xml-----
123 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
124   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
125 <hibernate-mapping>
126   <class name="Employee" table="employee">
127     <id name="employeeId" column="empno"/>
128     <property name="employeeName" column="ename"/>
129     <property name="employeeSal" column="sal"/>
130     <property name="deptNumber" column="deptno"/>
131   </class>
132 </hibernate-mapping>
133 -----InsertServlet.java-----
134 import java.io.*;
135 import javax.servlet.*;
136 import org.hibernate.*;
137 import org.hibernate.cfg.*;
138 public class InsertServlet extends GenericServlet
139 {
140     private SessionFactory factory;
141     //non-life cycle init method
142     public void init() throws ServletException
```

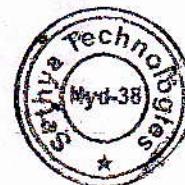


```

143     {
144         factory=new Configuration().configure().buildSessionFactory();
145     }
146     //life cycle service() method
147     public void service(ServletRequest req, ServletResponse res)
148             throws ServletException,IOException
149     {
150         //read input values
151         String s1 = req.getParameter("empno");
152         String s2 = req.getParameter("ename");
153         String s3 = req.getParameter("sal");
154         String s4 = req.getParameter("deptno");
155         //wrapping
156         int empno = Integer.parseInt(s1.trim());
157         int sal = Integer.parseInt(s3.trim());
158         int deptno = Integer.parseInt(s4.trim());
159         //open session
160         Session session = factory.openSession();
161         //create a pojo class object
162         Employee e = new Employee();
163         e.setEmployeeId(empno);
164         e.setEmployeeName(s2);
165         e.setEmployeeSal(sal);
166         e.setDeptNumber(deptno);
167         //begin transaction
168         Transaction tx = session.beginTransaction();
169         session.save(e);
170         tx.commit();
171         PrintWriter pw = res.getWriter();
172         pw.println("<h2> Employee Saved Successfully</h2> <br>");
173         pw.println("Return <a href=index.html>HOME</a>");
174         pw.close();
175         session.close();
176     } //service
177     public void destroy()
178     {
179         factory.close();
180     }
181 }

-----UpdateServlet.java-----
183 import java.io.*;
184 import javax.servlet.*;
185 import org.hibernate.*;
186 import org.hibernate.cfg.*;
187 public class UpdateServlet extends GenericServlet
188 {
189     private SessionFactory factory;
190     public void init() throws ServletException
191     {
192         factory=new Configuration().configure().buildSessionFactory();
193     }
194     public void service(ServletRequest req, ServletResponse res)
195             throws ServletException,IOException
196     {
197         //read input values
198         String s1 = req.getParameter("empno");
199         String s2 = req.getParameter("ename");
200         String s3 = req.getParameter("sal");
201         String s4 = req.getParameter("deptno");
202         int empno = Integer.parseInt(s1.trim());
203         int sal = Integer.parseInt(s3.trim());
204         int deptno = Integer.parseInt(s4.trim());
205         PrintWriter pw = res.getWriter();
206         Session session = factory.openSession();
207         Object o = session.get(Employee.class,empno);
208         if(o==null)
209         {
210             pw.println("<h2> Employee Number Not Found. So row");
211             pw.println("is not updated </h2> ");
212             pw.println("Return <a href=index.html>HOME</a> ");
213             return;

```



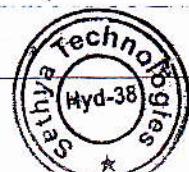
```
214         }
215         Employee e = (Employee)o;
216         Transaction tx = session.beginTransaction();
217         e.setEmployeeName(s2);
218         e.setEmployeeSal(sal);
219         e.setDeptNumber(deptno);
220         tx.commit();
221         pw.println("<h2> Employee updated Successfully</h2> <br>");
222         pw.println("Return <a href=index.html>HOME</a>");
223         pw.close();
224         session.close();
225     } //service
226     public void destroy()
227     {
228         factory.close();
229     }
230 }
231 -----SelectServlet.java-----
232 import java.io.*;
233 import javax.servlet.*;
234 import org.hibernate.*;
235 import org.hibernate.cfg.*;
236 public class SelectServlet extends GenericServlet
237 {
238     private SessionFactory factory;
239     public void init() throws ServletException
240     {
241         factory=new Configuration().configure().buildSessionFactory();
242     }
243     public void service(ServletRequest req, ServletResponse res)
244             throws ServletException, IOException
245     {
246         String s1 = req.getParameter("empno");
247         int empno = Integer.parseInt(s1.trim());
248         PrintWriter pw = res.getWriter();
249         Session session = factory.openSession();
250         Object o = session.get(Employee.class,empno);
251         if(o==null)
252         {
253             pw.println("<h2> Employee Number Not Found. So row is"
254             "not selected </h2>");
255             pw.println("Return <a href=index.html>HOME</a>");
256             return;
257         }
258         Employee e =(Employee)o;
259         pw.println("<h2>");
260         pw.println(e.getEmployeeId());
261         pw.println("<br>");
262         pw.println(e.getEmployeeName());
263         pw.println("<br>");
264         pw.println(e.getEmployeeSal());
265         pw.println("<br>");
266         pw.println(e.getDeptNumber());
267         pw.println("<br>");
268         pw.println("</h2>");
269         pw.println("Return <a href=index.html>HOME</a>");
270         pw.close();
271         session.close();
272     } //service
273     public void destroy()
274     {
275         factory.close();
276     }
277 }
278 -----DeleteServlet.java-----
279 import java.io.*;
280 import javax.servlet.*;
281 import org.hibernate.*;
282 import org.hibernate.cfg.*;
283 public class DeleteServlet extends GenericServlet
284 {
```



```

285     private SessionFactory factory;
286     public void init() throws ServletException
287     {
288         factory=new Configuration().configure().buildSessionFactory();
289     }
290     public void service(ServletRequest req, ServletResponse res)
291             throws ServletException,IOException
292     {
293         //read input values
294         String s1 = req.getParameter("empno");
295         int empno = Integer.parseInt(s1.trim());
296         PrintWriter pw = res.getWriter();
297         Session session = factory.openSession();
298         Object o = session.get(Employee.class,empno);
299         if(o==null)
300         {
301             pw.println("<h2> Employee Number Not Found. So row is
302             not deleted </h2>");
303             pw.println("Return <a href=index.html>HOME</a>");
304             return;
305         }
306         Transaction tx = session.beginTransaction();
307         session.delete(o);
308         tx.commit();
309         pw.println("<h2> Employee deleted Successfully</h2> <br>");
310         pw.println("Return <a href=index.html>HOME</a>");
311         pw.close();
312         session.close();
313     }//service
314     public void destroy()
315     {
316         factory.close();
317     }
318 }
319 -----
320 application-2(connecting with multiple databases from an application)
321 -----Student.java-----
322 public class Student
323 {
324     private Integer studentid;
325     private String studentname;
326     private Integer marks;
327     public Integer getMarks() {
328         return marks;
329     }
330     public void setMarks(Integer marks) {
331         this.marks = marks;
332     }
333     public Integer getStudentid() {
334         return studentid;
335     }
336     public void setStudentid(Integer studentid) {
337         this.studentid = studentid;
338     }
339     public String getStudentname() {
340         return studentname;
341     }
342     public void setStudentname(String studentname) {
343         this.studentname = studentname;
344     }
345 }
346 -----student.hbm.xml-----
347 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
348   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
349 <hibernate-mapping>
350     <class name="Student" table="student">
351         <id name="studentid" column="sid" type="java.lang.Integer"/>
352         <property name="studentname" column="sname" type="java.lang.String" length="10"/>
353         <property name="marks"/>
354     </class>
355 </hibernate-mapping>

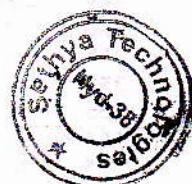
```



```

356 ----- hibernate_mysql.cfg.xml -----
357 <!DOCTYPE hibernate-configuration PUBLIC
358   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
359   "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
360 <hibernate-configuration>
361   <session-factory>
362     <!-- connection properties -->
363     <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
364     <property name="connection.url">jdbc:mysql://localhost:3306/test</property>
365     <property name="connection.username">root</property>
366     <property name="connection.password"></property>
367     <!-- hibernate properties -->
368     <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
369     <property name="show_sql">true</property>
370     <property name="hbm2ddl.auto">update</property>
371     <!-- mapping files -->
372     <mapping resource="student.hbm.xml"/>
373   </session-factory>
374 </hibernate-configuration>
375 ----- hibernate_postgresql.cfg.xml -----
376 <!DOCTYPE hibernate-configuration PUBLIC
377   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
378   "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
379 <!-- hibernate_postgresql.cfg.xml -->
380 <hibernate-configuration>
381   <session-factory>
382     <!-- connection properties -->
383     <property name="connection.driver_class">org.postgresql.Driver</property>
384     <property name="connection.url">jdbc:postgresql://localhost:5432/postgres</prop
385     <property name="connection.username">postgres</property>
386     <property name="connection.password">postgres</property>
387     <!-- hibernate properties -->
388     <property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>
389     <property name="show_sql">true</property>
390     <property name="hbm2ddl.auto">update</property>
391     <!-- mapping files -->
392     <mapping resource="student.hbm.xml"/>
393   </session-factory>
394 </hibernate-configuration>
395 ----- InsertClient.java -----
396 import org.hibernate.*;
397 import org.hibernate.cfg.*;
398 class InsertClient
399 {
400     public static void main(String[] args)
401     {
402         SessionFactory factory1 = new Configuration()
403             .configure("hibernate_mysql.cfg.xml").buildSessionFactory();
404         Session session1 = factory1.openSession();
405         SessionFactory factory2 = new Configuration()
406             .configure("hibernate_postgresql.cfg.xml").buildSessionFactory();
407         Session session2 = factory2.openSession();
408         Student s = new Student();
409         s.setStudentid(111);
410         s.setStudentname("aaa");
411         s.setMarks(200);
412         Transaction tx1 = session1.beginTransaction();
413         session1.persist(s);
414         tx1.commit();
415         Transaction tx2 = session2.beginTransaction();
416         session2.persist(s);
417         tx2.commit();
418         session1.close();
419         session2.close();
420         factory1.close();
421         factory2.close();
422     }
423 }
424 -----
425 ----- Application-3(Connection pooling with proxool) -----
426 ----- Product.java -----

```



```

427 public class Product
428 {
429     private int productId;
430     private String productName;
431     private double price;
432     public void setProductId(int productId)
433     {
434         this.productId = productId;
435     }
436     public int getProductId()
437     {
438         return productId;
439     }
440     public void setProductName(String productName)
441     {
442         this.productName = productName;
443     }
444     public String getProductName()
445     {
446         return productName;
447     }
448     public void setPrice(double price)
449     {
450         this.price=price;
451     }
452     public double getPrice()
453     {
454         return price;
455     }
456 }
-----product.hbm.xml-----
458 <!DOCTYPE hibernate-mapping PUBLIC
459   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
460   "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
461 <hibernate-mapping>
462   <class name="Product" table="product_info">
463     <id name="productId" column="pid" type="int"/>
464     <property name="productName" column="pname" length="10" type="string"/>
465     <property name="price" type="double"/>
466   </class>
467 </hibernate-mapping>
-----hibernate.cfg.xml-----
468 <!DOCTYPE hibernate-configuration PUBLIC
469   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
470   "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
471   <!-- hibernate.cfg.xml -->
472 <hibernate-configuration>
473   <session-factory>
474     <!-- proxool properties -->
475     <property name="hibernate.proxool.xml">proxool.xml</property>
476     <property name="hibernate.proxool.pool_alias">abcd</property>
477     <property name="hibernate.connection.provider_class">org.hibernate.
478                                         connection.ProxoolConnectionProvider</property>
479     <!-- hibernate properties -->
480     <property name="dialect">org.hibernate.dialect.OracleDialect</property>
481     <property name="show_sql">true</property>
482     <property name="hbm2ddl.auto">update</property>
483
484     <!-- mapping files -->
485     <mapping resource="product.hbm.xml"/>
486   </session-factory>
487 </hibernate-configuration>
-----proxool.xml-----
490 <!-- proxool.xml -->
491 <something>
492   <proxool>
493     <alias>abcd</alias>
494     <driver-url>jdbc:oracle:thin:@localhost:1521:satya</driver-url>
495     <driver-class>oracle.jdbc.OracleDriver</driver-class>
496     <driver-properties>
497       <property name="user" value="scott"/>

```



```

498         <property name="password" value="tiger"/>
499     </driver-properties>
500     <max-connections-count>5</max-connections-count>
501   </proxool>
502 </something>
503 -----ProductInsert.java-----
504 import org.hibernate.*;
505 import org.hibernate.cfg.*;
506 public class ProductInsert
507 {
508     public static void main(String args[ ])
509     {
510         Configuration conf = new Configuration();
511         conf.configure("hibernate.cfg.xml");
512         SessionFactory factory = conf.buildSessionFactory();
513         Session session = factory.openSession();
514         Product p = new Product();
515         p.setProductId(101);
516         p.setProductName("SONY");
517         p.setPrice(9000);
518         Transaction tx = session.beginTransaction();
519         session.save(p);
520         tx.commit();
521         session.close();
522         System.out.println("Product saved in Database successfully");
523         factory.close();
524     }
525 }
526 -----
527 Application-4(Server Connection pooling with Jndi)
528 -----Student.java-----
529 public class Student
530 {
531     private Integer studentid;
532     private String studentname;
533     private Integer marks;
534     public Integer getMarks() {
535         return marks;
536     }
537     public void setMarks(Integer marks) {
538         this.marks = marks;
539     }
540     public Integer getStudentid() {
541         return studentid;
542     }
543     public void setStudentid(Integer studentid) {
544         this.studentid = studentid;
545     }
546     public String getStudentname() {
547         return studentname;
548     }
549     public void setStudentname(String studentname) {
550         this.studentname = studentname;
551     }
552 }
553 -----student.hbm.xml-----
554 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
555 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
556 <!-- student.hbm.xml -->
557 <hibernate-mapping>
558   <class name="Student" table="student_info">
559     <id name="studentid" column="sid" type="java.lang.Integer">
560       <generator class="assigned"/>
561     </id>
562     <property name="studentname" column="sname" type="java.lang.String" length="10"/>
563     <property name="marks"/>
564   </class>
565 </hibernate-mapping>
566 -----hibernate.cfg.xml-----
567 <!DOCTYPE hibernate-configuration PUBLIC
568 "://Hibernate/Hibernate Configuration DTD 3.0//EN"

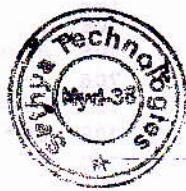
```



```

569         "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
570 <hibernate-configuration>
571     <session-factory>
572         <!-- jndi properties -->           JNDI
573         <property name="hibernate.connection.datasource">oraclejndi</property>
574         <property name="hibernate.jndi.class">com.sun.enterprise.naming.ldap.
575                                         SerialInitContextFactory</property>
576         <property name="hibernate.jndi.url">iip://localhost:4848</property>
577         <!-- hibernate properties -->
578         <property name="show_sql">true</property>
579         <property name="dialect">org.hibernate.dialect.OracleDialect</property>
580         <property name="hbm2ddl.auto">update</property>
581         <!-- mapping file -->
582         <mapping resource="student.hbm.xml"/>
583     </session-factory>
584 </hibernate-configuration>
585 -----Client.java-----
586 import org.hibernate.*;
587 import org.hibernate.cfg.*;
588 public class Client
589 {
590     public static void main(String args[])
591     {
592         Configuration conf = new Configuration();
593         conf.configure("hibernate.cfg.xml");
594         SessionFactory factory=conf.buildSessionFactory();
595         Session ses = factory.openSession();
596         Student s1 = new Student();
597         s1.setStudentid(102);
598         s1.setStudentname("XYZ");
599         s1.setMarks(600);
600         Transaction tx = ses.beginTransaction();
601         ses.save(s1);
602         tx.commit();
603         ses.close();
604         factory.close();
605     }
606 }
607 =====
608
609     Application-5(Table per class inheritance)
610 -----Payment.java-----
611 import java.util.*;
612 public class Payment
613 {
614     private int paymentId;
615     private double amount;
616     private Date paymentDate;
617     public double getAmount() {
618         return amount;
619     }
620     public void setAmount(double amount) {
621         this.amount = amount;
622     }
623     public int getPaymentId() {
624         return paymentId;
625     }
626     public void setPaymentId(int paymentId) {
627         this.paymentId = paymentId;
628     }
629     public void setPaymentDate(Date paymentDate)
630     {
631         this.paymentDate=paymentDate;
632     }
633     public Date getPaymentDate()
634     {
635         return paymentDate;
636     }
637 }
638 -----CreditCardPayment.java-----
639 public class CreditCardPayment extends Payment

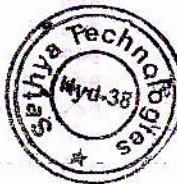
```



```

711      <!-- payment2.hbm.xml -->
712  <hibernate-mapping>
713    <class name="Payment" table="payments" schema="scott">
714      <id name="paymentId" column="payid">
715        <generator class="increment"/>
716      </id>
717      <property name="amount"/>
718      <property name="paymentDate" column="pdate"/>
719      <joined-subclass name="CreditCardPayment" table="credit_table">
720        <!-- foreign key mapping -->
721        <key column="pid"/>
722        <property name="cardType" column="cctype" length="8"/>
723      </joined-subclass>
724      <joined-subclass name="ChequePayment" table="cheque_table">
725        <key column="pid"/>
726        <property name="chequeType" column="chtype" length="8"/>
727      </joined-subclass>
728    </class>
729  </hibernate-mapping>
730  -----
731      Application-7(Table per concrete class)
732  -----payment3.hbm.xml-----
733  <!DOCTYPE hibernate-mapping PUBLIC
734    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
735    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
736  <!-- payment3.hbm.xml -->
737  <hibernate-mapping>
738    <class name="Payment">
739      <id name="paymentId" column="payid">
740        <generator class="increment"/>
741      </id>
742      <property name="amount"/>
743      <property name="paymentDate" column="pdate"/>
744      <union-subclass name="CreditCardPayment" table="credit_table">
745        <property name="cardType" column="cctype" length="8"/>
746      </union-subclass>
747      <union-subclass name="ChequePayment" table="cheque_table">
748        <property name="chequeType" column="chtype" length="8"/>
749      </union-subclass>
750    </class>
751  </hibernate-mapping>
752  -----
753      Application-8(HQL Select operation)
754  -----Employee.java-----
755 //Employee.java (POJO)
756 public class Employee
757 {
758     private int employeeId;
759     private String employeeName;
760     private int employeeSal;
761     private int deptNumber;
762     public int getDeptNumber() {
763         return deptNumber;
764     }
765     public void setDeptNumber(int deptNumber) {
766         this.deptNumber = deptNumber;
767     }
768     public String getEmployeeName() {
769         return employeeName;
770     }
771     public void setEmployeeName(String employeeName) {
772         this.employeeName = employeeName;
773     }
774     public int getEmployeeId() {
775         return employeeId;
776     }
777     public void setEmployeeId(int employeeId) {
778         this.employeeId = employeeId;
779     }
780     public int getEmployeeSal() {
781         return employeeSal;

```



```

782     }
783     public void setEmployeeSal(int employeeSal) {
784         this.employeeSal = employeeSal;
785     }
786 }
787 -----employee.hbm.xml-----
788 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
789   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
790 <!-- employee.hbm.xml -->
791 <hibernate-mapping>
792   <class name="Employee" table="employee">
793     <id name="employeeId" column="empno"/>
794     <property name="employeeName" column="ename"/>
795     <property name="employeeSal" column="sal"/>
796     <property name="deptNumber" column="deptno"/>
797   </class>
798 </hibernate-mapping>
799 -----SelectClient.java-----
800 //SelectClient.java
801 import java.util.*;
802 import org.hibernate.*;
803 import org.hibernate.cfg.*;
804 class SelectClient
805 {
806     public static void main(String[ ] args)
807     {
808         SessionFactory factory = new Configuration().configure().buildSessionFactory();
809         Session session = factory.openSession();
810         //way-1
811         Query qry1=session.createQuery("from Employee e where e.employeeSal>:p1");
812         qry1.setParameter("p1",2000);
813         List l1 = qry1.list();
814         Iterator it1 = l1.iterator();
815         while(it1.hasNext())
816         {
817             Employee e =(Employee)it1.next();
818             System.out.println(e.getEmployeeId()+"\t"+e.getEmployeeName()+"\t"
819                               +e.getEmployeeSal()+"\t"+e.getDeptNumber());
820         }
821         System.out.println("=====");
822         //way-2
823         Query qry2 = session.createQuery("select e.employeeId,e.employeeName from
824                                         Employee e where e.deptNumber=:p1");
825         qry2.setParameter("p1",30);
826         List l2 = qry2.list();
827         Iterator it2 = l2.iterator();
828         while(it2.hasNext())
829         {
830             Object r[ ] = (Object[ ])it2.next();
831             System.out.println(r[0]+"\t"+r[1]);
832         }
833         System.out.println("=====");
834         //way-3
835         Query qry3 = session.createQuery("select e.employeeName from Employee
836                                         where e.employeeName like '%$%' ");
837         List l3 = qry3.list();
838         Iterator it3 = l3.iterator();
839         while(it3.hasNext())
840         {
841             String s =(String)it3.next();
842             System.out.println(s);
843         }
844         System.out.println("=====");
845         session.close();
846         factory.close();
847     }
848 }
849 -----
850 Application-9(HQL Insert operation)
851 -----Employee.java-----
852 public class Employee

```



```

853 {
854     private int employeeId;
855     private String employeeName;
856     private int employeeSal;
857     private int deptNumber;
858     public int getDeptNumber() {
859         return deptNumber;
860     }
861     public void setDeptNumber(int deptNumber) {
862         this.deptNumber = deptNumber;
863     }
864     public String getEmployeeName() {
865         return employeeName;
866     }
867     public void setEmployeeName(String employeeName) {
868         this.employeeName = employeeName;
869     }
870     public int getEmployeeId() {
871         return employeeId;
872     }
873     public void setEmployeeId(int employeeId) {
874         this.employeeId = employeeId;
875     }
876     public int getEmployeeSal() {
877         return employeeSal;
878     }
879     public void setEmployeeSal(int employeeSal) {
880         this.employeeSal = employeeSal;
881     }
882 }
-----employee.hbm.xml-----
883 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
884      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
885 <!-- employee.hbm.xml -->
886 <hibernate-mapping>
887     <class name="Employee" table="employee">
888         <id name="employeeId" column="empno"/>
889         <property name="employeeName" column="ename"/>
890         <property name="employeeSal" column="sal"/>
891         <property name="deptNumber" column="deptno"/>
892     </class>
893 </hibernate-mapping>
894 -----
-----hibernate.cfg.xml-----
895 <!DOCTYPE hibernate-configuration PUBLIC
896      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
897      "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
898 <!-- hibernate.cfg.xml -->
899 <hibernate-configuration>
900     <session-factory>
901         <!-- connection properties -->
902         <property name="connection.driver_class">oracle.jdbc.OracleDriver</property>
903         <property name="connection.url">jdbc:oracle:thin:@localhost:1521:satya</property>
904         <property name="connection.username">scott</property>
905         <property name="connection.password">tiger</property>
906         <!-- hibernate properties -->
907         <property name="dialect">org.hibernate.dialect.OracleDialect</property>
908         <property name="show_sql">true</property>
909         <property name="hbm2ddl.auto">update</property>
910         <!-- mapping files -->
911         <mapping resource="employee.hbm.xml"/>
912         <mapping resource="testemployee.hbm.xml"/>
913     </session-factory>
914 </hibernate-configuration>
915 -----
-----TestEmployee.java-----
916 public class TestEmployee
917 {
918     private int empId;
919     private String empName;
920     private int empSal;
921     private int deptno;
922     public void setEmpId(int empId)

```



```

924     {
925         this.empId=empId;
926     }
927     public int getEmpId()
928     {
929         return empId;
930     }
931     public void setEmpName(String empName)
932     {
933         this.empName=empName;
934     }
935     public String getEmpName()
936     {
937         return empName;
938     }
939     public void setEmpSal(int empSal)
940     {
941         this.empSal=empSal;
942     }
943     public int getEmpSal()
944     {
945         return empSal;
946     }
947     public void setDeptno(int deptno)
948     {
949         this.deptno=deptno;
950     }
951     public int getDeptno()
952     {
953         return deptno;
954     }
955 };
956 -----testemployee.hbm.xml-----
957 <!-- testemployee.hbm.xml -->
958 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
959 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
960 <!-- testemployee.hbm.xml -->
961 <hibernate-mapping>
962     <class name="TestEmployee" table="TESTEMPLOYEE">
963         <id name="empId" column="empid"/>
964         <property name="empName" column="ename" length="12"/>
965         <property name="empSal" column="sal"/>
966         <property name="deptno"/>
967     </class>
968 </hibernate-mapping>
969 -----InsertClient.java-----
970 import org.hibernate.*;
971 import org.hibernate.cfg.*;
972 public class InsertClient
973 {
974     public static void main(String s[])
975     {
976         SessionFactory factory = new Configuration().configure().buildSessionFactory();
977         Session session = factory.openSession();
978         Query qry = session.createQuery("insert into TestEmployee(empId,empName,
979         empSal,deptno) select e.employeeId,e.employeeName,e.employeeSal,e.deptNumber
980             from Employee e");
981         Transaction tx = session.beginTransaction();
982         int k = qry.executeUpdate();
983         tx.commit();
984         System.out.println(k+" objects inserted");
985         Query qry1=session.createQuery("update Employee e set e.employeeSal=9000
986             where e.deptNumber=30");
987         Transaction tx1 = session.beginTransaction();
988         int k1 = qry1.executeUpdate();
989         tx1.commit();
990         System.out.println(k1+" objects are updated");
991         session.close();
992         factory.close();
993     }
994 }

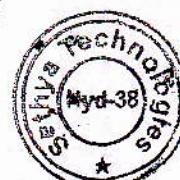
```



```

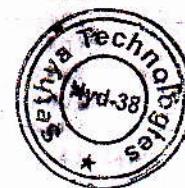
995 -----
996     Application-10(Criteria Test)
997 -----Employee.java-----
998 public class Employee {
999 {
1000     private int employeeId;
1001     private String employeeName;
1002     private int employeeSal;
1003     private int deptNumber;
1004     public int getDeptNumber() {
1005         return deptNumber;
1006     }
1007     public void setDeptNumber(int deptNumber) {
1008         this.deptNumber = deptNumber;
1009     }
1010     public String getEmployeeName() {
1011         return employeeName;
1012     }
1013     public void setEmployeeName(String employeeName) {
1014         this.employeeName = employeeName;
1015     }
1016     public int getEmployeeId() {
1017         return employeeId;
1018     }
1019     public void setEmployeeId(int employeeId) {
1020         this.employeeId = employeeId;
1021     }
1022     public int getEmployeeSal() {
1023         return employeeSal;
1024     }
1025     public void setEmployeeSal(int employeeSal) {
1026         this.employeeSal = employeeSal;
1027     }
1028 }
1029 -----employee.hbm.xml-----
1030 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
1031      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
1032 <!-- employee.hbm.xml -->
1033 <hibernate-mapping>
1034     <class name="Employee" table="employee">
1035         <id name="employeeId" column="empno"/>
1036         <property name="employeeName" column="ename"/>
1037         <property name="employeeSal" column="sal"/>
1038         <property name="deptNumber" column="deptno"/>
1039     </class>
1040 </hibernate-mapping>
1041 -----CritClient.java-----
1042 import org.hibernate.*;
1043 import org.hibernate.cfg.*;
1044 import org.hibernate.criterion.*;
1045 import java.util.*;
1046 class CritClient
1047 {
1048     public static void main(String[] args)
1049     {
1050         SessionFactory factory = new Configuration().configure().buildSessionFactory();
1051         Session session = factory.openSession();
1052         //loading full objects(way1)
1053         Criteria crit = session.createCriteria(Employee.class);
1054         //condition is empname should contain 'S' and order is ascending
1055         Criterion condition1 = Restrictions.ilike("employeeName", "%s%");
1056         crit.add(condition1);
1057         crit.addOrder(Order.asc("employeeName"));
1058         List l = crit.list();
1059         //for each loop
1060         for(Object o : l)
1061         {
1062             Employee e=(Employee)o;
1063             System.out.println(e.getEmployeeId()+" "+e.getEmployeeName()+
1064                         " "+e.getEmployeeSal()+" "+e.getDeptNumber());
1065         }
}

```



```

1066     System.out.println("=====");
1067     //loading partial properties of objects(way2)
1068     Criteria crit2 = session.createCriteria(Employee.class);
1069     Projection p1 = Projections.property("employeeName");
1070     Projection p2 = Projections.property("employeeSal");
1071     ProjectionList plist = Projections.projectionList();
1072     plist.add(p1);
1073     plist.add(p2);
1074     crit2.setProjection(plist);
1075     List l2 = crit2.list();
1076     Iterator it2 = l2.iterator();
1077     while(it2.hasNext())
1078     {
1079         Object obj[] =(Object[ ])it2.next();
1080         System.out.println(obj[0]+"" " +obj[1]);
1081     }
1082     System.out.println("=====");
1083     //loading objects with single property(way3)
1084     Criteria crit3 = session.createCriteria(Employee.class);
1085     Projection p3 = Projections.property("employeeSal");
1086     crit3.setProjection(p3);
1087     List l3 = crit3.list();
1088     Iterator it3 = l3.iterator();
1089     while(it3.hasNext())
1090     {
1091         Integer i =(Integer)it3.next();
1092         System.out.println(i);
1093     }
1094     System.out.println("=====");
1095     session.close();
1096     factory.close();
1097 }
1098 }
1099 -----
1100 Application-11(Native SQL application)
1101 -----Employee.java-----
1102 public class Employee
1103 {
1104     private int employeeId;
1105     private String employeeName;
1106     private int employeeSal;
1107     private int deptNumber;
1108     public int getDeptNumber() {
1109         return deptNumber;
1110     }
1111     public void setDeptNumber(int deptNumber) {
1112         this.deptNumber = deptNumber;
1113     }
1114     public String getEmployeeName() {
1115         return employeeName;
1116     }
1117     public void setEmployeeName(String employeeName) {
1118         this.employeeName = employeeName;
1119     }
1120     public int getEmployeeId() {
1121         return employeeId;
1122     }
1123     public void setEmployeeId(int employeeId) {
1124         this.employeeId = employeeId;
1125     }
1126     public int getEmployeeSal() {
1127         return employeeSal;
1128     }
1129     public void setEmployeeSal(int employeeSal) {
1130         this.employeeSal = employeeSal;
1131     }
1132 }
1133 -----employee.hbm.xml-----
1134 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
1135     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
1136 <hibernate-mapping>
```



```

1137 <class name="Employee" table="employee">
1138   <id name="employeeId" column="empno"/>
1139   <property name="employeeName" column="ename"/>
1140   <property name="employeeSal" column="sal"/>
1141   <property name="deptNumber" column="deptno"/>
1142 </class>
1143 </hibernate-mapping>
-----NativeClient.java-----
1145 import org.hibernate.*;
1146 import org.hibernate.cfg.*;
1147 import java.util.*;
1148 class NativeClient
1149 {
1150     public static void main(String[] args)
1151     {
1152         SessionFactory factory = new Configuration().configure().buildSessionFactory();
1153         Session session = factory.openSession();
1154         //non-select
1155         SQLQuery qry1 = session.createSQLQuery("insert into employee
1156                                         values(?, ?, ?, ?, ?)");
1157         qry1.setParameter(0, 7799);
1158         qry1.setParameter(1, "ROM");
1159         qry1.setParameter(2, 5000);
1160         qry1.setParameter(3, 10);
1161         Transaction tx = session.beginTransaction();
1162         int k = qry1.executeUpdate();
1163         tx.commit();
1164         System.out.println(k+" row inserted....");
1165         System.out.println("=====");
1166         //select operation
1167         SQLQuery qry2 = session.createSQLQuery("select * from employee
1168                                         where deptno=10");
1169         qry2.addEntity(Employee.class); //Entity query object
1170         List l = qry2.list();
1171         Iterator it = l.iterator();
1172         while(it.hasNext())
1173         {
1174             Employee e = (Employee)it.next();
1175             System.out.println(e.getEmployeeId()+"\t"+e.getEmployeeName()+"\t"+
1176                               e.getEmployeeSal()+"\t"+e.getDeptNumber());
1177         }
1178         session.close();
1179         factory.close();
1180     }
1181 }
-----Application-12(calling a function using Hibernate)-----
1182 -----Employee.java-----
1183 public class Employee
1184 {
1185     private int employeeId;
1186     private String employeeName;
1187     private int employeeSal;
1188     private int deptNumber;
1189     public int getDeptNumber() {
1190         return deptNumber;
1191     }
1192     public void setDeptNumber(int deptNumber) {
1193         this.deptNumber = deptNumber;
1194     }
1195     public String getEmployeeName() {
1196         return employeeName;
1197     }
1198     public void setEmployeeName(String employeeName) {
1199         this.employeeName = employeeName;
1200     }
1201     public int getEmployeeId() {
1202         return employeeId;
1203     }
1204     public void setEmployeeId(int employeeId) {
1205         this.employeeId = employeeId;
1206     }

```



```

1208     }
1209     public int getEmployeeSal() {
1210         return employeeSal;
1211     }
1212     public void setEmployeeSal(int employeeSal) {
1213         this.employeeSal = employeeSal;
1214     }
1215 }
-----employee.hbm.xml-----
1216 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
1217   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
1218 <hibernate-mapping>
1219   <class name="Employee" table="employee">
1220     <id name="employeeId" column="empno"/>
1221     <property name="employeeName" column="ename"/>
1222     <property name="employeeSal" column="sal"/>
1223     <property name="deptNumber" column="deptno"/>
1224   </class>
1225   <sql-query name="s1" callable="true">
1226     <return class="Employee" alias="e"/>
1227     {?=?call selectAllEmployees()}
1228   </sql-query>
1229 </hibernate-mapping>
1230 -----client.java-----
1231 import org.hibernate.*;
1232 import org.hibernate.cfg.*;
1233 import java.util.*;
1234 public class client
1235 {
1236     public static void main(String args[])
1237     {
1238         SessionFactory factory = new Configuration().configure().buildSessionFactory();
1239         Session ses = factory.openSession();
1240         Query qry = ses.getNamedQuery("s1");
1241         List l = qry.list();
1242         Iterator it = l.iterator();
1243         while(it.hasNext())
1244         {
1245             Employee e =(Employee)it.next();
1246             System.out.println(e.getEmployeeId()+"\t"+e.getEmployeeName()+"\t"+
1247                               e.getEmployeeSal()+"\t"+e.getDeptNumber());
1248             System.out.println("=====");
1249         }
1250     }
1251     ses.close();
1252     factory.close();
1253 }
1254 }
-----Application-13(Pagination Application using Servlet with Hibernate)-----
1255 -----Employee.java-----
1256 public class Employee
1257 {
1258     private int employeeId;
1259     private String employeeName;
1260     private int employeeSal;
1261     private int deptNumber;
1262     public int getDeptNumber() {
1263         return deptNumber;
1264     }
1265     public void setDeptNumber(int deptNumber) {
1266         this.deptNumber = deptNumber;
1267     }
1268     public String getEmployeeName() {
1269         return employeeName;
1270     }
1271     public void setEmployeeName(String employeeName) {
1272         this.employeeName = employeeName;
1273     }
1274     public int getEmployeeId() {
1275         return employeeId;
1276     }
1277 }

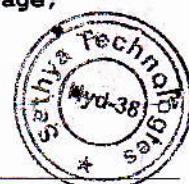
```



```

1279     public void setEmployeeId(int employeeId) {
1280         this.employeeId = employeeId;
1281     }
1282     public int getEmployeeSal() {
1283         return employeeSal;
1284     }
1285     public void setEmployeeSal(int employeeSal) {
1286         this.employeeSal = employeeSal;
1287     }
1288 }
-----employee.hbm.xml-----
1289 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
1290   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
1291 <hibernate-mapping>
1292   <class name="Employee" table="employee">
1293     <id name="employeeId" column="empno"/>
1294     <property name="employeeName" column="ename"/>
1295     <property name="employeeSal" column="sal"/>
1296     <property name="deptNumber" column="deptno"/>
1297   </class>
1298 </hibernate-mapping>
1300 -----web.xml-----
1301 <web-app>
1302   <servlet>
1303     <servlet-name>paginationtest</servlet-name>
1304     <servlet-class>PaginationServlet</servlet-class>
1305   </servlet>
1306   <servlet-mapping>
1307     <servlet-name>paginationtest</servlet-name>
1308     <url-pattern>/ps</url-pattern>
1309   </servlet-mapping>
1310 </web-app>
-----PaginationServlet.java-----
1311 import java.io.*;
1312 import javax.servlet.*;
1313 import java.util.*;
1314 import org.hibernate.*;
1315 import org.hibernate.cfg.*;
1316 import org.hibernate.criterion.*;
1317 public class PaginationServlet extends GenericServlet
1318 {
1319     int pageIndex=0;
1320     SessionFactory factory;
1321     public void init() throws ServletException
1322     {
1323         factory = new Configuration().configure().buildSessionFactory();
1324         System.out.println("factory created.....");
1325     }
1326     public void service(ServletRequest req, ServletResponse res)
1327             throws ServletException, IOException
1328     {
1329         int totalNumberOfRecords=0;
1330         int numberOfRowsPerPage=3;
1331         //logic-1
1332         String sPageIndex = req.getParameter("pageindex");
1333         if(sPageIndex==null)
1334         {
1335             pageIndex=1;
1336         }
1337         else
1338         {
1339             pageIndex=Integer.parseInt(sPageIndex);
1340         }
1341         Session session = factory.openSession();
1342         //logic-2
1343         int si=(pageIndex*numberOfRecordsPerPage) - numberOfRowsPerPage;
1344         Criteria crit2 = session.createCriteria(Employee.class);
1345         crit2.setFirstResult(si);
1346         crit2.setMaxResults(numberOfRecordsPerPage);
1347         List l2 = crit2.list();
1348         Iterator it2 = l2.iterator();
1349

```



```

1350     PrintWriter pw = res.getWriter();
1351     pw.println("<center><table border=2 width='80%' height='70%'>");
1352     pw.println("<tr>");
1353     pw.println("<th>EMPNO</th><th>ENAME</th><th>SAL</th><th>DEPTNO</th>");
1354     pw.println("</tr>");
1355     while(it2.hasNext())
1356     {
1357         Employee e =(Employee)it2.next();
1358         pw.println("<tr>");
1359         pw.println("<td>"+e.getEmployeeId()+"</td>");
1360         pw.println("<td>"+e.getEmployeeName()+"</td>");
1361         pw.println("<td>"+e.getEmployeeSal()+"</td>");
1362         pw.println("<td>"+e.getDeptNumber()+"</td>");
1363         pw.println("</tr>");
1364     }//while
1365     pw.println("</table>");
1366     //logic-3
1367     Criteria crit1 =session.createCriteria(Employee.class);
1368     crit1.setProjection(Projections.rowCount());
1369     List l1 = crit1.list();
1370     Iterator it1 = l1.iterator();
1371     if(it1.hasNext())
1372     {
1373         Object o = it1.next();
1374         totalNumberOfRecords=Integer.parseInt(o.toString());
1375     }
1376     //logic-4
1377     int noOfPages = totalNumberOfRecords/numberOfRecordsPerPage;
1378     if(totalNumberOfRecords > (noOfPages * numberOfRecordsPerPage))
1379     {
1380         noOfPages = noOfPages+1;
1381     }
1382     pw.println("<br>");
1383     //logic-5
1384     pw.println("<table width=70% height=20%>");
1385     pw.println("<tr>");
1386     pw.println("<td>");
1387     if(pageIndex>1)
1388     {
1389
1390         pw.println("<a href=ps?pageindex="+ (pageIndex-1)+"> Prev </a>");}
1391     }
1392     else
1393     {
1394         pw.println("nbsp;");
1395     }
1396     pw.println("</td>");pw.println("nbsp; </td>");pw.println("nbsp;");pw.println("<td>");if(pageIndex <noOfPages)
1400     {
1401         pw.println("<a href=ps?pageindex="+(pageIndex+1)+"> Next </a>");}
1402     }
1403     else
1404     {
1405         pw.println("nbsp;");}
1406     pw.println("</td>");pw.println("</table>");pw.println("</center>");session.close();pw.close();
1407     }
1408     public void destroy()
1409     {
1410         factory.close();System.out.println("factory closed");
1411     }
1412     }
1413     public void destroy()
1414     {
1415         factory.close();System.out.println("factory closed");
1416     }
1417     }
1418 }
```



sekar-hibernate-examples-2.txt

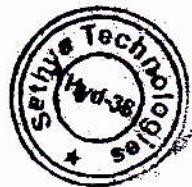
## Application-14(One-To-Many Example)

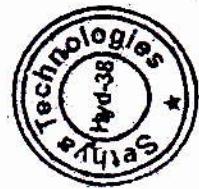
## -----Vendor.java-----

```

1 //Parent POJO
2 //Vendor.java
3 import java.util.*;
4 public class Vendor
5 {
6     private int vendorId;
7     private String vendorName;
8     private Set customers;
9     public Set getCustomers() {
10         return customers;
11     }
12     public void setCustomers(Set customers) {
13         this.customers = customers;
14     }
15     public int getVendorId() {
16         return vendorId;
17     }
18     public void setVendorId(int vendorId) {
19         this.vendorId = vendorId;
20     }
21     public String getVendorName() {
22         return vendorName;
23     }
24     public void setVendorName(String vendorName) {
25         this.vendorName = vendorName;
26     }
27 }
28 -----Customer.java-----
29 //Child POJO
30 //Customer.java
31 public class Customer
32 {
33     private int customerId;
34     private String customerName;
35     private String customerAddress;
36     public int getCustomerId() {
37         return customerId;
38     }
39     public void setCustomerId(int customerId) {
40         this.customerId = customerId;
41     }
42     public String getCustomerName() {
43         return customerName;
44     }
45     public void setCustomerName(String customerName) {
46         this.customerName = customerName;
47     }
48     public void setCustomerAddress(String customerAddress)
49     {
50         this.customerAddress=customerAddress;
51     }
52     public String getCustomerAddress()
53     {
54         return customerAddress;
55     }
56 }
57 -----vendor.hbm.xml-----
58 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
59   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
60 <!-- vendor.hbm.xml -->
61 <hibernate-mapping>
62   <class name="Vendor" table="vendor">
63     <id name="vendorId" column="vid"/>
64     <property name="vendorName" column="vname" length="10"/>
65     <set name="customers" cascade="all" lazy="true">
66       <key column="venid"/>
67     </set>
68   </class>
69 </hibernate-mapping>

```





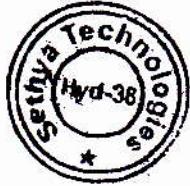
```
1 <one-to-many class="Customer"/>
2   <set>
3   </class>
4 </hibernate-mapping>
5 -----customer.hbm.xml-----
6 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
7   "http://.hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
8 <!-- customer.hbm.xml -->
9 <hibernate-mapping>
10 <class name="Customer" table="customer">
11   <id name="customerId" column="custid"/>
12   <property name="customerName" column="custname" length="10"/>
13   <property name="customerAddress" column="custadd" length="10"/>
14 </class>
15 </hibernate-mapping>
16 -----InsertClient.java-----
17 import java.util.*;
18 import org.hibernate.*;
19 import org.hibernate.cfg.*;
20 public class InsertClient
21 {
22     public static void main(String args[])
23     {
24         SessionFactory factory = new Configuration().configure().buildSessionFactory();
25         Session session = factory.openSession();
26         //Parent object
27         Vendor v = new Vendor();
28         v.setVendorId(111);
29         v.setVendorName("IBM");
30         //Child object-1
31         Customer c1 = new Customer();
32         c1.setCustomerId(501);
33         c1.setCustomerName("INFY");
34         c1.setCustomerAddress("HYD");
35         //Child object -2
36         Customer c2 = new Customer();
37         c2.setCustomerId(502);
38         c2.setCustomerName("TCS");
39         c2.setCustomerAddress("HYD");
40         //Child object -3
41         Customer c3 = new Customer();
42         c3.setCustomerId(503);
43         c3.setCustomerName("VERIZON");
44         c3.setCustomerAddress("US");
45         //step1
46         Set s = new HashSet();
47         s.add(c1);
48         s.add(c2);
49         s.add(c3);
50         //step2
51         v.setCustomers(s);
52         Transaction tx = session.beginTransaction();
53         session.save(v);
54         tx.commit();
55         session.close();
56         factory.close();
57     }
58 }
59 -----AdditionalChild.java-----
60 import java.util.*;
61 import org.hibernate.*;
62 import org.hibernate.cfg.*;
63 public class AdditionalChild
64 {
65     public static void main(String args[])
66     {
67         SessionFactory factory = new Configuration().configure().buildSessionFactory();
68         Session session = factory.openSession();
```



```

137     //step-1
138     Vendor v =(Vendor)session.get(Vendor.class,111);
139     //step-2
140     Set s = v.getCustomers();
141     //step-3
142     Customer c4 = new Customer();
143     c4.setCustomerId(504);
144     c4.setCustomerName("CTS");
145     c4.setCustomerAddress("US");
146     //step-4
147     Transaction tx = session.beginTransaction();
148     s.add(c4);
149     tx.commit();
150     session.close();
151     factory.close();
152   }
153 }
-----SelectClient.java-----
154 import java.util.*;
155 import org.hibernate.*;
156 import org.hibernate.cfg.*;
157 public class SelectClient
158 {
159   public static void main(String args[])
160   {
161     SessionFactory factory = new Configuration().configure().buildSessionFactory();
162     Session session = factory.openSession();
163     Vendor v= (Vendor)session.get(Vendor.class,111);
164     System.out.println("Vendor name = "+v.getVendorName());
165     Set s = v.getCustomers();
166     Iterator it = s.iterator();
167     while(it.hasNext())
168     {
169       Object o = it.next();
170       Customer c =(Customer)o;
171       System.out.println(c.getCustomerId()+" "+c.getCustomerName()
172                     +" "+c.getCustomerAddress());
173     }
174   }
175   session.close();
176   factory.close();
177 }
178 }
-----DeleteClient.java-----
179 import org.hibernate.*;
180 import org.hibernate.cfg.*;
181 public class DeleteClient
182 {
183   public static void main(String args[])
184   {
185     SessionFactory factory = new Configuration().configure().buildSessionFactory();
186     Session session = factory.openSession();
187     Vendor v=(Vendor)session.get(Vendor.class,111);
188     Transaction tx = session.beginTransaction();
189     session.delete(v);
190     tx.commit();
191     session.close();
192     factory.close();
193   }
194 }
195 }
196 =====
197 Application-15(Many-To-One Example)
198 -----Vendor.java-----
199 //Parent POJO
200 //Vendor.java
201 public class Vendor
202 {
203   private int vendorId;
204   private String vendorName;

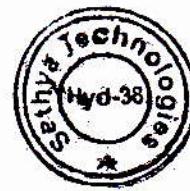
```



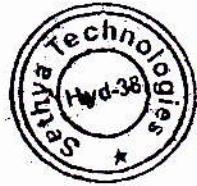
```
205     public int getVendorId() {
206         return vendorId;
207     }
208     public void setVendorId(int vendorId) {
209         this.vendorId = vendorId;
210     }
211     public String getVendorName() {
212         return vendorName;
213     }
214     public void setVendorName(String vendorName) {
215         this.vendorName = vendorName;
216     }
217 }
218 -----Customer.java-----
219 //Child POJO
220 //Customer.java
221 public class Customer {
222 {
223     private int customerId;
224     private String customerName;
225     private String customerAddress;
226     private Vendor vendor;
227     public void setVendor(Vendor vendor)
228     {
229         this.vendor=vendor;
230     }
231     public Vendor getVendor()
232     {
233         return vendor;
234     }
235     public int getCustomerId() {
236         return customerId;
237     }
238     public void setCustomerId(int customerId) {
239         this.customerId = customerId;
240     }
241     public String getCustomerName() {
242         return customerName;
243     }
244     public void setCustomerName(String customerName) {
245         this.customerName = customerName;
246     }
247     public void setCustomerAddress(String customerAddress)
248     {
249         this.customerAddress=customerAddress;
250     }
251     public String getCustomerAddress()
252     {
253         return customerAddress;
254     }
255 }
256 -----vendor.hbm.xml-----
257 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
258   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
259 <!-- vendor.hbm.xml -->
260 <hibernate-mapping>
261   <class name="Vendor" table="vendor">
262     <id name="vendorId" column="vid">
263       <generator class="assigned"/>
264     </id>
265     <property name="vendorName" column="vname" length="10"/>
266   </class>
267 </hibernate-mapping>
268 -----customer.hbm.xml-----
269 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
270   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
271 <!-- customer.hbm.xml -->
272 <hibernate-mapping>
```



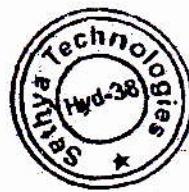
```
273 <class name="Customer" table="customer">
274   <id name="customerId" column="custid">
275     <generator class="assigned"/>
276   </id>
277   <property name="customerName" column="custname" length="10"/>
278   <property name="customerAddress" column="custadd" length="10"/>
279   <many-to-one name="vendor" class="Vendor" column="venid" cascade="all"
280           lazy="proxy"/>
281 </class>
282 </hibernate-mapping>
283 -----InsertClient.java-----
284 import org.hibernate.*;
285 import org.hibernate.cfg.*;
286 public class InsertClient
287 {
288   public static void main(String args[])
289   {
290     SessionFactory factory = new Configuration().configure().buildSessionFactory();
291     Session session = factory.openSession();
292     //Parent object
293     Vendor v = new Vendor();
294     v.setVendorId(111);
295     v.setVendorName("IBM");
296     //Child object-1
297     Customer c1 = new Customer();
298     c1.setCustomerId(501);
299     c1.setCustomerName("INFY");
300     c1.setCustomerAddress("HYD");
301     //Child object -2
302     Customer c2 = new Customer();
303     c2.setCustomerId(502);
304     c2.setCustomerName("TCS");
305     c2.setCustomerAddress("HYD");
306     //Child object -3
307     Customer c3 = new Customer();
308     c3.setCustomerId(503);
309     c3.setCustomerName("VERIZON");
310     c3.setCustomerAddress("US");
311     //add parent object to child objects
312     c1.setVendor(v);
313     c2.setVendor(v);
314     c3.setVendor(v);
315     Transaction tx = session.beginTransaction();
316     session.save(c1);
317     session.save(c2);
318     session.save(c3);
319     tx.commit();
320     session.close();
321     factory.close();
322   }
323 }
324 -----SelectClient.java-----
325 import org.hibernate.*;
326 import org.hibernate.cfg.*;
327 public class SelectClient
328 {
329   public static void main(String args[])
330   {
331     SessionFactory factory = new Configuration().configure().buildSessionFactory();
332     Session session = factory.openSession();
333     Customer c =(Customer)session.get(Customer.class,501);
334     System.out.println(c.getCustomerName()+" "+c.getCustomerAddress());
335     Vendor v = c.getVendor();
336     System.out.println(v.getVendorId()+" "+v.getVendorName());
337     session.close();
338     factory.close();
339   }
340 }
```



```
341 -----DeleteClient.java-----
342 import org.hibernate.*;
343 import org.hibernate.cfg.*;
344 public class DeleteClient
345 {
346     public static void main(String args[])
347     {
348         SessionFactory factory = new Configuration().configure().buildSessionFactory();
349         Session session = factory.openSession();
350         Customer c =(Customer)session.get(Customer.class,501);
351         Transaction tx = session.beginTransaction();
352         session.delete(c);
353         tx.commit();
354         session.close();
355         factory.close();
356     }
357 }
358 =====
359 Application-16(One-To-Many Bi-directional Example)
360 -----Vendor.java-----
361 //Parent POJO
362 //Vendor.java
363 import java.util.*;
364 public class Vendor
365 {
366     private int vendorId;
367     private String vendorName;
368     private Set customers;
369     public Set getCustomers()
370     {
371         return customers;
372     }
373     public void setCustomers(Set customers)
374     {
375         this.customers = customers;
376     }
377     public int getVendorId()
378     {
379         return vendorId;
380     }
381     public void setVendorId(int vendorId)
382     {
383         this.vendorId = vendorId;
384     }
385     public String getVendorName()
386     {
387         return vendorName;
388     }
389     public void setVendorName(String vendorName)
390     {
391         this.vendorName = vendorName;
392     }
393 }
394 -----Customer.java-----
395 //Child POJO
396 //Customer.java
397 public class Customer
398 {
399     private int customerId;
400     private String customerName;
401     private String customerAddress;
402     private Vendor vendor;
403     public void setVendor(Vendor vendor)
404     {
405         this.vendor=vendor;
406     }
407     public Vendor getVendor()
408     {
409         return vendor;
410     }
411     public int getCustomerId()
412     {
413         return customerId;
414     }
415     public void setCustomerId(int customerId)
416     {
417 }
```



```
409     this.customerId = customerId;
410 }
411 public String getCustomerName() {
412     return customerName;
413 }
414 public void setCustomerName(String customerName) {
415     this.customerName = customerName;
416 }
417 public void setCustomerAddress(String customerAddress)
418 {
419     this.customerAddress=customerAddress;
420 }
421 public String getCustomerAddress()
422 {
423     return customerAddress;
424 }
425 }
426 -----vendor.hbm.xml-----
427 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
428   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
429 <!-- vendor.hbm.xml -->
430 <hibernate-mapping>
431   <class name="Vendor" table="vendor">
432     <id name="vendorId" column="vid">
433       <generator class="assigned"/>
434     </id>
435     <property name="vendorName" column="vname" length="10"/>
436     <set name="customers" cascade="all" lazy="true" inverse="true">
437       <key column="venid"/>
438       <one-to-many class="Customer"/>
439     </set>
440   </class>
441 </hibernate-mapping>
442 -----
443 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
444   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
445 <!-- customer.hbm.xml -->
446 <hibernate-mapping>
447   <class name="Customer" table="customer">
448     <id name="customerId" column="custid">
449       <generator class="assigned"/>
450     </id>
451     <property name="customerName" column="custname" length="10"/>
452     <property name="customerAddress" column="custadd" length="10"/>
453     <many-to-one name="vendor" class="Vendor" column="venid" cascade="all"
454       lazy="proxy"/>
455   </class>
456 </hibernate-mapping>
457 -----
458 import java.util.*;
459 import org.hibernate.*;
460 import org.hibernate.cfg.*;
461 public class InsertClient
462 {
463     public static void main(String args[])
464     {
465         SessionFactory factory = new Configuration().configure().buildSessionFactory();
466         Session session = factory.openSession();
467         //Parent object
468         Vendor v = new Vendor();
469         v.setVendorId(111);
470         v.setVendorName("IBM");
471         //Child object-1
472         Customer c1 = new Customer();
473         c1.setCustomerId(501);
474         c1.setCustomerName("INFY");
475         c1.setCustomerAddress("HYD");
476         //Child object -2
```



```
477     Customer c2 = new Customer();
478     c2.setCustomerId(502);
479     c2.setCustomerName("TCS");
480     c2.setCustomerAddress("HYD");
481     //Child object -3
482     Customer c3 = new Customer();
483     c3.setCustomerId(503);
484     c3.setCustomerName("VERIZON");
485     c3.setCustomerAddress("US");
486     //many-to-one
487     c1.setVendor(v);
488     c2.setVendor(v);
489     c3.setVendor(v);
490     //one-to-many
491     //step1
492     Set s = new HashSet();
493     s.add(c1);
494     s.add(c2);
495     s.add(c3);
496     //step2
497     v.setCustomers(s);
498     Transaction tx = session.beginTransaction();
499     session.save(c1); // session.save(v)
500     tx.commit();
501     session.close();
502     factory.close();
503   }
504 }
```

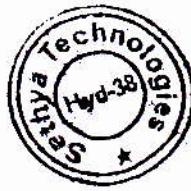
---

**506 Application-17(Many-To-Many Example)**

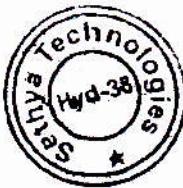
---

**-----Student.java-----**

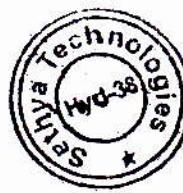
```
508 import java.util.*;
509 public class Student {
510   {
511     private int studentId;
512     private String studentName;
513     private String studentGrp;
514     private Set courses;
515     public Set getCourses() {
516       return courses;
517     }
518     public void setCourses(Set courses) {
519       this.courses = courses;
520     }
521     public String getStudentGrp() {
522       return studentGrp;
523     }
524     public void setStudentGrp(String studentGrp) {
525       this.studentGrp = studentGrp;
526     }
527     public int getStudentId() {
528       return studentId;
529     }
530     public void setStudentId(int studentId) {
531       this.studentId = studentId;
532     }
533     public String getStudentName() {
534       return studentName;
535     }
536     public void setStudentName(String studentName) {
537       this.studentName = studentName;
538     }
539   }
540 -----Course.java-----
541 import java.util.*;
542 public class Course {
543   {
544     private int courseid;
```



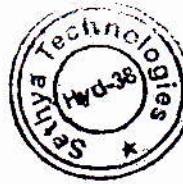
```
545     private String courseName;
546     private int duration;
547     private Set students;
548     public int getCourseld() {
549         return courseld;
550     }
551     public void setCourseld(int courseld) {
552         this.courseld = courseld;
553     }
554     public String getCourseName() {
555         return courseName;
556     }
557     public void setCourseName(String courseName) {
558         this.courseName = courseName;
559     }
560     public int getDuration() {
561         return duration;
562     }
563     public void setDuration(int duration) {
564         this.duration = duration;
565     }
566     public Set getStudents() {
567         return students;
568     }
569     public void setStudents(Set students) {
570         this.students = students;
571     }
572 }
573 -----student.hbm.xml-----
574 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
575   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
576 <!-- student.hbm.xml -->
577 <hibernate-mapping>
578   <class name="Student" table="student">
579     <id name="studentId" column="sid"/>
580     <property name="studentName" column="sname" length="10"/>
581     <property name="studentGrp" column="grp" length="10"/>
582     <set name="courses" table="students_courses" cascade="all">
583       <key column="sid_fk"/>
584       <many-to-many class="Course" column="cid_fk"/>
585     </set>
586   </class>
587 </hibernate-mapping>
588 -----course.hbm.xml-----
589 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
590   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
591 <!-- course.hbm.xml -->
592 <hibernate-mapping>
593   <class name="Course" table="course">
594     <id name="courseld" column="cid"/>
595     <property name="courseName" column="cname" length="10"/>
596     <property name="duration"/>
597     <set name="students" table="students_courses" cascade="all">
598       <key column="cid_fk"/>
599       <many-to-many class="Student" column="sid_fk"/>
600     </set>
601   </class>
602 </hibernate-mapping>
603 -----InsertClient.java-----
604 import org.hibernate.*;
605 import org.hibernate.cfg.*;
606 import java.util.*;
607 public class InsertClient
608 {
609     public static void main(String args[ ])
610     {
611         SessionFactory factory=new Configuration().configure().buildSessionFactory();
612         Session session = factory.openSession();
```



```
613     //Student obj--1
614     Student s1 = new Student();
615     s1.setStudentId(1001);
616     s1.setStudentName("aaa");
617     s1.setStudentGrp("BTech");
618     //Student obj -- 2
619     Student s2=new Student();
620     s2.setStudentId(1002);
621     s2.setStudentName("bbb");
622     s2.setStudentGrp("MCA");
623     //Course obj – 1
624     Course c1 = new Course();
625     c1.setCourseId(901);
626     c1.setCourseName("java");
627     c1.setDuration(150);
628     //Course obj – 2
629     Course c2 = new Course();
630     c2.setCourseId(902);
631     c2.setCourseName("oracle");
632     c2.setDuration(100);
633     Set set = new HashSet();
634     set.add(c1);
635     set.add(c2);
636     s1.setCourses(set);
637     s2.setCourses(set);
638     Transaction tx = session.beginTransaction();
639     session.save(s1);
640     session.save(s2);
641     tx.commit();
642     session.close();
643     factory.close();
644 }
645 }
646 -----AdditionalCourse.java-----
647 import org.hibernate.*;
648 import org.hibernate.cfg.*;
649 import java.util.*;
650 public class AdditionalCourse
651 {
652     public static void main(String args[ ])
653     {
654         SessionFactory factory=new Configuration().configure().buildSessionFactory();
655         Session session = factory.openSession();
656         //step-1
657         Student s=(Student)session.get(Student.class,1001);
658         //step-2
659         Set set = s.getCourses();
660         //step-3
661         Course c3 = new Course();
662         c3.setCourseId(903);
663         c3.setCourseName("Cpp");
664         c3.setDuration(50);
665         //step-4
666         Transaction tx = session.beginTransaction();
667         set.add(c3);
668         tx.commit();
669         session.close();
670         factory.close();
671     }
672 }
673 -----DeleteCourse.java-----
674 import org.hibernate.*;
675 import org.hibernate.cfg.*;
676 import java.util.*;
677 public class DeleteCourse
678 {
679     public static void main(String args[ ])
680     {
```



```
681 SessionFactory factory=new Configuration().configure().buildSessionFactory();
682 Session session = factory.openSession();
683 //step-1
684 Student s=(Student)session.get(Student.class,1001);
685 //step-2
686 Set set = s.getCourses();
687 //step-3
688 Course c =(Course)session.get(Course.class,902);
689 //step-4
690 Transaction tx = session.beginTransaction();
691 set.remove(c);
692 tx.commit();
693 session.close();
694 factory.close();
695 }
696 }
697 =====
698 Application-18(One-To-One with Foreign Key Example)
699 -----Person.java-----
700 //Person.java
701 public class Person
702 {
703     private int personId;
704     private String personName;
705     public void setPersonId(int personId)
706     {
707         this.personId=personId;
708     }
709     public int getPersonId()
710     {
711         return personId;
712     }
713     public void setPersonName(String personName)
714     {
715         this.personName=personName;
716     }
717     public String getPersonName()
718     {
719         return personName;
720     }
721 };
722 -----License.java-----
723 //License.java
724 import java.util.*;
725 public class License
726 {
727     private int licensId;
728     private Date issuedDate;
729     private Date expireDate;
730     private Person person;
731     public void setLicensId(int licensId)
732     {
733         this.licensId=licensId;
734     }
735     public int getLicensId()
736     {
737         return licensId;
738     }
739     public void setIssuedDate(Date issuedDate)
740     {
741         this.issuedDate=issuedDate;
742     }
743     public Date getIssuedDate()
744     {
745         return issuedDate;
746     }
747     public void setExpireDate(Date expireDate)
748     {
```

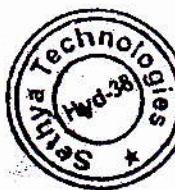


```

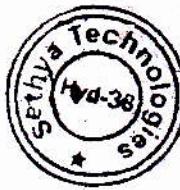
    this.expireDate=expireDate;
}
public Date getExpireDate()
{
    return expireDate;
}
public void setPerson(Person person)
{
    this.person=person;
}
public Person getPerson()
{
    return person;
}
}

-----person.hbm.xml-----
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- person.hbm.xml -->
<hibernate-mapping>
    <class name="Person" table="person">
        <id name="personId" column="pid"/>
        <property name="personName" column="pname" length="10"/>
    </class>
</hibernate-mapping>
-----license.hbm.xml-----
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- license.hbm.xml -->
<hibernate-mapping>
    <class name="License" table="license">
        <id name="licenseId" column="lid"/>
        <property name="issuedDate" column="idate"/>
        <property name="expireDate" column="edate"/>
        <many-to-one name="person" class="Person"
            column="per_id" unique="true" not-null="true" cascade="all"/>
    </class>
</hibernate-mapping>
-----InsertClient.java-----
import java.util.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
class InsertClient
{
    public static void main(String[] args)
    {
        SessionFactory factory= new Configuration().configure().buildSessionFactory();
        Session session = factory.openSession();
        //Person object
        Person p = new Person();
        p.setPersonId(1101);
        p.setPersonName("ABC");
        //License object
        License l1= new License();
        l1.setLicenseId(1010);
        l1.setIssuedDate(new Date());
        l1.setExpireDate(new Date());
        /* License l2= new License();
        l2.setLicenseId(1011);
        l2.setIssuedDate(new Date());
        l2.setExpireDate(new Date()); */
        l1.setPerson(p);
        // l2.setPerson(p);
        Transaction tx = session.beginTransaction();
        session.save(l1);
        //session.save(l2);
        tx.commit();
        session.close();
    }
}

```



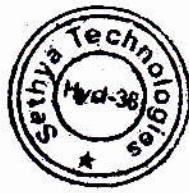
```
817     factory.close();
818 }
819 }
=====
821 Application-19(One-To-One with Primary key)
822 -----Person.java-----
823 //Person.java
824 public class Person
825 {
826     private int personId;
827     private String personName;
828     public void setPersonId(int personId)
829     {
830         this.personId=personId;
831     }
832     public int getPersonId()
833     {
834         return personId;
835     }
836     public void setPersonName(String personName)
837     {
838         this.personName=personName;
839     }
840     public String getPersonName()
841     {
842         return personName;
843     }
844 };
845 -----License.java-----
846 //License.java
847 import java.util.*;
848 public class License
849 {
850     private int licenseld;
851     private Date issuedDate;
852     private Date expireDate;
853     private Person person;
854     public void setLicenseld(int licenseld)
855     {
856         this.licenseld=licenseld;
857     }
858     public int getLicenseld()
859     {
860         return licenseld;
861     }
862     public void setIssuedDate(Date issuedDate)
863     {
864         this.issuedDate=issuedDate;
865     }
866     public Date getIssuedDate()
867     {
868         return issuedDate;
869     }
870     public void setExpireDate(Date expireDate)
871     {
872         this.expireDate=expireDate;
873     }
874     public Date getExpireDate()
875     {
876         return expireDate;
877     }
878     public void setPerson(Person person)
879     {
880         this.person=person;
881     }
882     public Person getPerson()
883     {
884         return person;
```



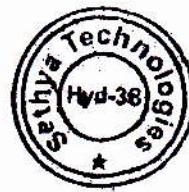
```

935     }
936
937 -----person.hbm.xml-----
938 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
939   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
940 <!-- person.hbm.xml -->
941 <hibernate-mapping>
942   <class name="Person" table="person">
943     <id name="personId" column="pid"/>
944       <property name="personName" column="pname" length="10"/>
945   </class>
946 </hibernate-mapping>
947 -----license.hbm.xml-----
948 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
949   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
950 <!-- license.hbm.xml -->
951 <hibernate-mapping>
952   <class name="License" table="license">
953     <id name="licenseId" column="lid">
954       <generator class="foreign">
955         <param name="property">person</param>
956       </generator>
957     </id>
958     <property name="issuedDate" column="idate"/>
959     <property name="expireDate" column="edate"/>
960     <one-to-one name="person" class="Person"
961       cascade="all"/>
962   </class>
963 </hibernate-mapping>
964 -----InsertClient.java-----
965 import java.util.*;
966 import org.hibernate.*;
967 import org.hibernate.cfg.*;
968 class InsertClient
969 {
970   public static void main(String[] args)
971   {
972     SessionFactory factory= new Configuration().configure().buildSessionFactory();
973     Session session = factory.openSession();
974     //Person object
975     Person p = new Person();
976     p.setPersonId(1101);
977     p.setPersonName("ABC");
978     //License object
979     License l = new License();
980     l.setIssuedDate(new Date());
981     l.setExpireDate(new Date());
982     l.setPerson(p);
983     Transaction tx = session.beginTransaction();
984     session.save(l);
985     tx.commit();
986     session.close();
987     factory.close();
988   }
989 }
990 -----
991 Application-20(component mapping example)
992 -----Person.java-----
993 //Person.java
994 import java.util.*;
995 public class Person
996 {
997   private int personId;
998   private PName pname;
999   private Date dob;
1000  public void setPersonId(int personId)
1001  {
1002    this.personId=personId;

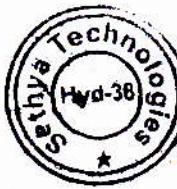
```



```
953     }
954     public int getPersonId()
955     {
956         return personId;
957     }
958     public void setPname(PName pname)
959     {
960         this.pname=pname;
961     }
962     public PName getPname()
963     {
964         return pname;
965     }
966     public void setDob(Date dob)
967     {
968         this.dob=dob;
969     }
970     public Date getDob()
971     {
972         return dob;
973     }
974 };
975 -----PName.java-----
976 //PName.java
977 public class PName
978 {
979     private char initial;
980     private String firstName;
981     private String lastName;
982     public void setInitial(char initial)
983     {
984         this.initial=initial;
985     }
986     public char getInitial()
987     {
988         return initial;
989     }
990     public void setFirstName(String firstName)
991     {
992         this.firstName=firstName;
993     }
994     public String getFirstName()
995     {
996         return firstName;
997     }
998     public void setLastName(String lastName)
999     {
1000        this.lastName=lastName;
1001    }
1002     public String getLastName()
1003     {
1004         return lastName;
1005     }
1006 }
1007 -----person.hbm.xml-----
1008 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
1009      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
1010 <!-- person.hbm.xml -->
1011 <hibernate-mapping>
1012   <class name="Person" table="persons">
1013     <id name="personId" column="pid"/>
1014     <property name="dob" column="dob"/>
1015     <component name="pname">
1016       <property name="initial" column="init" length="5"/>
1017       <property name="firstName" column="fname" length="10"/>
1018       <property name="lastName" column="lname" length="10"/>
1019     </component>
1020   </class>
```



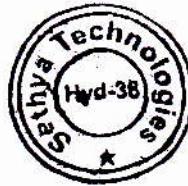
```
1021 </hibernate-mapping>
1022 -----
1023 -----Client.java-----
1024 import java.util.*;
1025 import org.hibernate.*;
1026 import org.hibernate.cfg.*;
1027 class Client
1028 {
1029     public static void main(String[ ] args)
1030     {
1031         SessionFactory factory = new Configuration().configure().buildSessionFactory();
1032         Session session = factory.openSession();
1033         PName p = new PName();
1034         p.setInitial('S');
1035         p.setFirstName("Abc");
1036         p.setLastName("Xyz");
1037         Person pr = new Person();
1038         pr.setPersonId(111);
1039         pr.setDob(new Date());
1040         //assign PName object to Person
1041         pr.setPname(p);
1042         Transaction tx = session.beginTransaction();
1043         session.save(pr);
1044         tx.commit();
1045         session.close();
1046         factory.close();
1047     }
1048 }
1049 =====
1050 Application-21(A First example using Annotations)
1051 -----Student.java-----
1052 //Student.java
1053 import javax.persistence.*;
1054 @Entity
1055 @Table(name="student_table")
1056 public class Student
1057 {
1058     @Id
1059     @Column(name="stno")
1060     private Integer sno;
1061     @Column(name="sname",length=10)
1062     private String sname;
1063     @Column(name="stadd",length=10)
1064     private String address;
1065     public void setAddress(String address) {
1066         this.address = address;
1067     }
1068     public String getAddress() {
1069         return address;
1070     }
1071     public void setSname(String sname) {
1072         this.sname = sname;
1073     }
1074     public String getSname() {
1075         return sname;
1076     }
1077     public void setSno(Integer sno) {
1078         this.sno = sno;
1079     }
1080     public Integer getSno() {
1081         return sno;
1082     }
1083 }
1084 -----hibernate.cfg.xml-----
1085 <!DOCTYPE hibernate-configuration PUBLIC
1086     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
1087     "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
1088 <hibernate-configuration>
```



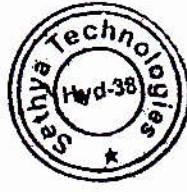
```

1089 <session-factory>
1090   <property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
1091   <property name="hibernate.connection.url">|dbc:oracle:thin:@localhost:1521:satya</property>
1092   <property name="hibernate.connection.username">scott</property>
1093   <property name="hibernate.connection.password">tiger</property>
1094   <property name="show_sql">true </property>
1095   <property name="hbm2ddl.auto">update</property>
1096   <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
1097   <mapping class="Student"/>
1098 </session-factory>
1099 </hibernate-configuration>
1100 =====InsertClient.java=====
1101 import org.hibernate.*;
1102 import org.hibernate.cfg.*;
1103 public class InsertClient
1104 {
1105     public static void main(String args[])
1106     {
1107         SessionFactory factory=new Configuration().configure().buildSessionFactory();
1108         Session ses = factory.openSession();
1109         Student s = new Student();
1110         s.setSno(101);
1111         s.setSname("ssreddy");
1112         s.setAddress("hyd");
1113         Transaction tx = ses.beginTransaction();
1114         ses.persist(s);
1115         tx.commit();
1116         ses.close();
1117         factory.close();
1118     }
1119 }
1120 =====
1121 Application-22(Table per class using annotations)
1122 -----Payment.java-----
1123 import javax.persistence.*;
1124 @Entity
1125 @Table(name="PAYMENT")
1126 @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
1127 @DiscriminatorColumn(name="pmode",discriminatorType=DiscriminatorType.STRING,length=4)
1128 public class Payment
1129 {
1130     @Id
1131     @Column(name="payid")
1132     private Integer paymentId;
1133     @Column(name="amount")
1134     private Double amount;
1135     public Double getAmount() {
1136         return amount;
1137     }
1138     public void setAmount(Double amount) {
1139         this.amount = amount;
1140     }
1141     public Integer getPaymentId() {
1142         return paymentId;
1143     }
1144     public void setPaymentId(Integer paymentId) {
1145         this.paymentId = paymentId;
1146     }
1147 }
1148 -----ChequePayment.java-----
1149 //ChequePayment.java
1150 import javax.persistence.*;
1151 @Entity
1152 @DiscriminatorValue(value="CH")
1153 public class ChequePayment extends Payment
1154 {
1155     @Column(name="ctype",length=10)
1156     private String chequeType;

```



```
1157     public String getChequeType() {
1158         return chequeType;
1159     }
1160     public void setChequeType(String chequeType) {
1161         this.chequeType = chequeType;
1162     }
1163 }
-----CreditCardPayment.java-----
1164 //CreditCardPayment.java
1165 import javax.persistence.*;
1166 @Entity
1167 @DiscriminatorValue(value="CR")
1168 public class CreditCardPayment extends Payment
1169 {
1170     @Column(name="cctype",length=10)
1171     private String cardType;
1172     public String getCardType() {
1173         return cardType;
1174     }
1175     public void setCardType(String cardType) {
1176         this.cardType = cardType;
1177     }
1178 }
-----hibernate.cfg.xml-----
1181 <!DOCTYPE hibernate-configuration PUBLIC
1182     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
1183     "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
1184 <hibernate-configuration>
1185 <session-factory>
1186 <property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
1187 <property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:satya</property>
1188 <property name="hibernate.connection.username">scott</property>
1189 <property name="hibernate.connection.password">tiger</property>
1190 <property name="show_sql">true </property>
1191 <property name="hbm2ddl.auto">update</property>
1192 <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
1193 <mapping class="Payment"/>
1194 <mapping class="CreditCardPayment"/>
1195 <mapping class="ChequePayment"/>
1196 </session-factory>
1197 </hibernate-configuration>
-----InsertClient.java-----
1198 import org.hibernate.*;
1199 import org.hibernate.cfg.*;
1200 public class InsertClient
1201 {
1202     public static void main(String[] args)
1203     {
1204         SessionFactory factory = new Configuration().configure().buildSessionFactory();
1205         Session session=factory.openSession();
1206         CreditCardPayment ccp = new CreditCardPayment();
1207         ccp.setPaymentId(101);
1208         ccp.setAmount(5000.0);
1209         ccp.setCardType("VISA");
1210         ChequePayment cp = new ChequePayment();
1211         cp.setPaymentId(102);
1212         cp.setAmount(8000.0);
1213         cp.setChequeType("ORDER");
1214         Transaction tx = session.beginTransaction();
1215         session.save(ccp);
1216         session.save(cp);
1217         tx.commit();
1218         session.close();
1219         factory.close();
1220     }
1221 }
-----Application-23(Table per subclass using annotations)-----
```



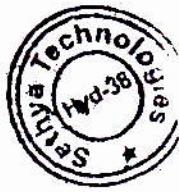
```

1255 -----Payment.java-----
1256 //Payment.java
1257 import javax.persistence.*;
1258 @Entity
1259 @Table(name="PAYMENT")
1260 @Inheritance(strategy=InheritanceType.JOINED)
1261 public class Payment
1262 {
1263     @Id
1264     @Column(name="payid")
1265     private Integer paymentId;
1266     @Column(name="amount")
1267     private Double amount;
1268     public Double getAmount() {
1269         return amount;
1270     }
1271     public void setAmount(Double amount) {
1272         this.amount = amount;
1273     }
1274     public Integer getPaymentId() {
1275         return paymentId;
1276     }
1277     public void setPaymentId(Integer paymentId) {
1278         this.paymentId = paymentId;
1279     }
1280 }
1281 -----CreditCardPayment.java-----
1282 //CreditCardPayment.java
1283 import javax.persistence.*;
1284 @Entity
1285 @Table(name="CREDIT_TABLE")
1286 @PrimaryKeyJoinColumn(name="pid")
1287 public class CreditCardPayment extends Payment
1288 {
1289     @Column(name="cctype",length=10)
1290     private String cardType;
1291     public String getCardType() {
1292         return cardType;
1293     }
1294     public void setCardType(String cardType) {
1295         this.cardType = cardType;
1296     }
1297 }
1298 -----ChequePayment.java-----
1299 //ChequePayment.java
1300 import javax.persistence.*;
1301 @Entity
1302 @Table(name="CHEQUE_TABLE")
1303 @PrimaryKeyJoinColumn(name="p_id")
1304 public class ChequePayment extends Payment
1305 {
1306     @Column(name="ctype",length=10)
1307     private String chequeType;
1308     public String getChequeType() {
1309         return chequeType;
1310     }
1311     public void setChequeType(String chequeType) {
1312         this.chequeType = chequeType;
1313     }
1314 }
1315 -----InsertClient.java-----
1316 import org.hibernate.*;
1317 import org.hibernate.cfg.*;
1318 public class InsertClient
1319 {
1320     public static void main(String[] args)
1321     {
1322         SessionFactory factory = new Configuration().configure().buildSessionFactory();

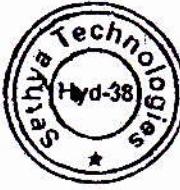
```



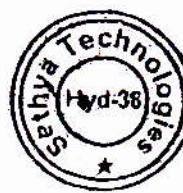
```
1293     Session session=factory.openSession();
1294     CreditCardPayment ccp = new CreditCardPayment();
1295     ccp.setPaymentId(101);
1296     ccp.setAmount(5000.0);
1297     ccp.setCardType("VISA");
1298     ChequePayment cp = new ChequePayment();
1299     cp.setPaymentId(102);
1300     cp.setAmount(8000.0);
1301     cp.setChequeType("ORDER");
1302     Transaction tx = session.beginTransaction();
1303     session.save(ccp);
1304     session.save(cp);
1305     tx.commit();
1306     session.close();
1307     factory.close();
1308 }
1309 =====
1310 Application-24(One-To-Many Relationship using Annotations)
1311 -----Vendor.java-----
1312 //Parent POJO
1313 //Vendor.java
1314 import javax.persistence.*;
1315 import java.util.*;
1316 @Entity
1317 @Table(name="vendor")
1318 public class Vendor
1319 {
1320     @Id
1321     @Column(name="vid")
1322     private int vendorId;
1323     @Column(name="vname",length=10)
1324     private String vendorName;
1325     @OneToMany(targetEntity=Customer.class,cascade=CCascadeType.ALL)
1326     @JoinColumn(name="venid",referencedColumnName="vid")
1327     private Set customers;
1328     public Set getCustomers() {
1329         return customers;
1330     }
1331     public void setCustomers(Set customers) {
1332         this.customers = customers;
1333     }
1334     public int getVendorId() {
1335         return vendorId;
1336     }
1337     public void setVendorId(int vendorId) {
1338         this.vendorId = vendorId;
1339     }
1340     public String getVendorName() {
1341         return vendorName;
1342     }
1343     public void setVendorName(String vendorName) {
1344         this.vendorName = vendorName;
1345     }
1346 }
1347 }
1348 -----Customer.java-----
1349 //Child POJO
1350 //Customer.java
1351 import javax.persistence.*;
1352 @Entity
1353 @Table(name="customer")
1354 public class Customer
1355 {
1356     @Id
1357     @Column(name="custid")
1358     private int customerId;
1359     @Column(name="custname",length=10)
1360     private String custName;
```



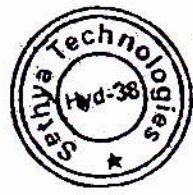
```
1355  
1356  
1357  
1358  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1380  
1381  
1382  
1383  
1384 }  
----- hibernate.cfg.xml -----  
1385 <!DOCTYPE hibernate-configuration PUBLIC  
1386 " -//Hibernate/Hibernate Configuration DTD 3.0//EN"  
1387 " http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd ">  
1388 <!-- hibernate.cfg.xml -->  
1389 <hibernate-configuration>  
1390   <session-factory>  
1391     <!-- connection properties -->  
1392     <property name="connection.driver_class">oracle.jdbc.OracleDriver</property>  
1393     <property name="connection.url">jdbc:oracle:thin:@localhost:1521:satya</property>  
1394     <property name="connection.username">scott</property>  
1395     <property name="connection.password">tiger</property>  
1396     <!-- hibernate properties -->  
1397     <property name="dialect">org.hibernate.dialect.OracleDialect</property>  
1398     <property name="show_sql">true</property>  
1399     <property name="hbm2ddl.auto">update</property>  
1400     <!-- mapping classes -->  
1401     <mapping class="Vendor"/>  
1402     <mapping class="Customer"/>  
1403   </session-factory>  
1404 </hibernate-configuration>  
----- InsertClient.java -----  
1405 import java.util.*;  
1406 import org.hibernate.*;  
1407 import org.hibernate.cfg.*;  
1408 public class InsertClient  
1409 {  
1410   public static void main(String args[ ])  
1411   {  
1412     SessionFactory factory = new Configuration().configure().buildSessionFactory();  
1413     Session session = factory.openSession();  
1414     //Parent object  
1415     Vendor v = new Vendor();  
1416     v.setVendorId(111);  
1417     v.setVendorName("IBM");  
1418     //Child object-1  
1419     Customer c1 = new Customer();  
1420     c1.setCustomerId(501);  
1421     c1.setCustomerName("INFY");  
1422     c1.setCustomerAddress("HYD");  
1423     //Child object -2  
1424     Customer c2 = new Customer();  
1425     c2.setCustomerId(502);  
1426     c2.setCustomerName("TCS");  
1427  
1428 }
```



```
1429     c2.setCustomerAddress("HYD");
1430     //Child object -3
1431     Customer c3 = new Customer();
1432     c3.setCustomerId(503);
1433     c3.setCustomerName("VERIZON");
1434     c3.setCustomerAddress("US");
1435     //step1
1436     Set s = new HashSet();
1437     s.add(c1);
1438     s.add(c2);
1439     s.add(c3);
1440     //step2
1441     v.setCustomers(s);
1442     Transaction tx = session.beginTransaction();
1443     session.save(v);
1444     tx.commit();
1445     session.close();
1446     factory.close();
1447 }
1448 }
=====
1450 Application-25(Many-To-One using Annotations)
1451 -----Vendor.java-----
1452 //Vendor.java
1453 import javax.persistence.*;
1454 @Entity
1455 @Table(name="vendor")
1456 public class Vendor
{
1457
1458     @Id
1459     @Column(name="vid")
1460     private int vendorId;
1461     @Column(name="vname",length=10)
1462     private String vendorName;
1463     public int getVendorId() {
1464         return vendorId;
1465     }
1466     public void setVendorId(int vendorId) {
1467         this.vendorId = vendorId;
1468     }
1469     public String getVendorName() {
1470         return vendorName;
1471     }
1472     public void setVendorName(String vendorName) {
1473         this.vendorName = vendorName;
1474     }
1475 }
1476 -----Customer.java-----
1477 //Customer.java
1478 import javax.persistence.*;
1479 @Entity
1480 @Table(name="customer")
1481 public class Customer
{
1482
1483     @Id
1484     @Column(name="custid")
1485     private int customerId;
1486     @Column(name="custname",length=10)
1487     private String customerName;
1488     @Column(name="custadd",length=10)
1489     private String customerAddress;
1490     @ManyToOne(targetEntity=Vendor.class,cascade=CascadeType.ALL,fetch=FetchType.EAGER)
1491     @JoinColumn(name="venid")
1492     private Vendor vendor;
1493     public void setVendor(Vendor vendor)
1494     {
1495         this.vendor=vendor;
1496     }
```



```
1501     public Vendor getVendor()
1502     {
1503         return vendor;
1504     }
1505     public int getCustomerId() {
1506         return customerId;
1507     }
1508     public void setCustomerId(int customerId) {
1509         this.customerId = customerId;
1510     }
1511     public String getCustomerName() {
1512         return customerName;
1513     }
1514     public void setCustomerName(String customerName) {
1515         this.customerName = customerName;
1516     }
1517     public void setCustomerAddress(String customerAddress)
1518     {
1519         this.customerAddress=customerAddress;
1520     }
1521 }
1522 -----hibernate.cfg.xml-----
1523 <!DOCTYPE hibernate-configuration PUBLIC
1524     "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
1525     "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
1526 <!-- hibernate.cfg.xml -->
1527 <hibernate-configuration>
1528     <session-factory>
1529         <!-- connection properties -->
1530         <property name="connection.driver_class">oracle.jdbc.OracleDriver</property>
1531         <property name="connection.url">jdbc:oracle:thin:@localhost:1521:satya</property>
1532         <property name="connection.username">scott</property>
1533         <property name="connection.password">tiger</property>
1534         <!-- hibernate properties -->
1535         <property name="dialect">org.hibernate.dialect.OracleDialect</property>
1536         <property name="show_sql">true</property>
1537         <property name="hbm2ddl.auto">update</property>
1538         <!-- mapping classes -->
1539         <mapping class="Vendor"/>
1540         <mapping class="Customer"/>
1541     </session-factory>
1542 </hibernate-configuration>
1543 -----InsertClient.java-----
1544 import org.hibernate.*;
1545 import org.hibernate.cfg.*;
1546 public class InsertClient
1547 {
1548     public static void main(String args[])
1549     {
1550         SessionFactory factory = new Configuration().configure().buildSessionFactory();
1551         Session session = factory.openSession();
1552         //Parent object
1553         Vendor v = new Vendor();
1554         v.setVendorId(111);
1555         v.setVendorName("IBM");
1556         //Child object-1
1557         Customer c1 = new Customer();
1558         c1.setCustomerId(501);
1559         c1.setCustomerName("INFY");
1560         c1.setCustomerAddress("HYD");
1561         //Child object -2
1562         Customer c2 = new Customer();
1563         c2.setCustomerId(502);
1564         c2.setCustomerName("TCS");
```



```

1565      c2.setCustomerAddress("HYD");
1566      //Child object -3
1567      Customer c3 = new Customer();
1568      c3.setCustomerId(503);
1569      c3.setCustomerName("VERIZON");
1570      c3.setCustomerAddress("US");
1571      //add parent object to child objects
1572      c1.setVendor(v);
1573      c2.setVendor(v);
1574      c3.setVendor(v);
1575      Transaction tx = session.beginTransaction();
1576      session.save(c1);
1577      session.save(c2);
1578      session.save(c3);
1579      tx.commit();
1580      session.close();
1581      factory.close();
1582  }
1583 }
=====
```

#### Application-26(Many-To-Many using Annotations)

---

//Item.java

```

1587 import javax.persistence.*;
1588 import java.util.*;
1589 @Entity
1590 @Table(name="items")
1591 public class Item
1592 {
1593     @Id
1594     @Column(name="item_id")
1595     private int itemId;
1596     @Column(name="item_name",length=10)
1597     private String itemName;
1598     @ManyToMany(targetEntity=Category.class,mappedBy="items")
1599     private Set categories;
1600     public Set getCategories() {
1601         return categories;
1602     }
1603     public void setCategories(Set categories) {
1604         this.categories = categories;
1605     }
1606     public int getItemId() {
1607         return itemId;
1608     }
1609     public void setItemId(int itemId) {
1610         this.itemId = itemId;
1611     }
1612     public String getItemName() {
1613         return itemName;
1614     }
1615     public void setItemName(String itemName) {
1616         this.itemName = itemName;
1617     }
1618 }
1619 }
```

---

-----Category.java-----

//Category.java

```

1621 import javax.persistence.*;
1622 import java.util.*;
1623 @Entity
1624 @Table(name="categories")
1625 public class Category
1626 {
1627     @Id
1628     @Column(name="cat_id")
1629     private int categoryId;
1630     @Column(name="cat_name",length=10)
1631     private String categoryName;
```



```
1633     @ManyToMany(targetEntity=Item.class,cascade=CascadeType.ALL)
1634     @JoinTable(name="categories_items",
1635         joinColumns=@JoinColumn(name="cat_id_fk",referencedColumnName="cat_id"),
1636         inverseJoinColumns=@JoinColumn(name="item_id_fk",referencedColumnName="item_id"))
1637     private Set items;
1638     public int getCategoryid() {
1639         return categoryId;
1640     }
1641     public void setCategoryid(int categoryId) {
1642         this.categoryId = categoryId;
1643     }
1644     public Set getItems() {
1645         return items;
1646     }
1647     public void setItems(Set items) {
1648         this.items = items;
1649     }
1650     public String getCategoryName() {
1651         return categoryName;
1652     }
1653     public void setCategoryName(String categoryname) {
1654         this.categoryName = categoryname;
1655     }
1656 }
1657 -----hibernate.cfg.xml-----
1658 <!DOCTYPE hibernate-configuration PUBLIC
1659   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
1660   "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
1661 <hibernate-configuration>
1662 <session-factory>
1663 <property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
1664 <property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:satya</property>
1665 <property name="hibernate.connection.username">scott</property>
1666 <property name="hibernate.connection.password">tiger</property>
1667 <property name="show_sql">true </property>
1668 <property name="hbm2ddl.auto">update</property>
1669 <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
1670 <mapping class="Category"/>
1671 <mapping class="Item"/>
1672 </session-factory>
1673 </hibernate-configuration>
1674 -----InsertClient.java-----
1675 import org.hibernate.*;
1676 import org.hibernate.cfg.*;
1677 import java.util.*;
1678 public class InsertClient {
1679     public static void main(String args[])
1680     {
1681         SessionFactory factory=new Configuration().configure().buildSessionFactory();
1682         Session ses = factory.openSession();
1683         Category c1 = new Category();
1684         c1.setCategoryId(1);
1685         c1.setCategoryName("aaa");
1686         Category c2=new Category();
1687         c2.setCategoryId(2);
1688         c2.setCategoryName("bbb");
1689         Item i1=new Item();
1690         i1.setItemId(101);
1691         i1.setItemName("item1");
1692         Item i2 = new Item();
1693         i2.setItemId(102);
1694         i2.setItemName("item2");
1695         Set s =new HashSet();
1696         s.add(i1);
1697         s.add(i2);
1698         c1.setItems(s);
1699         c2.setItems(s);
1700         Transaction tx = ses.beginTransaction();
```



```
1701     ses.save(c1);
1702     ses.save(c2);
1703     tx.commit();
1704     ses.close();
1705   }
1706 =====
1707   Application-27(One-To-One using Annotations)
1708   -----
1709   Person.java
1710
1711  //Person.java
1712  import javax.persistence.*;
1713  @Entity
1714  @Table(name="person")
1715  public class Person
1716  {
1717     @Id
1718     @Column(name="per_id")
1719     private int personId;
1720     @Column(name="per_name",length=10)
1721     private String personName;
1722     public void setPersonId(int personId)
1723     {
1724         this.personId=personId;
1725     }
1726     public int getPersonId()
1727     {
1728         return personId;
1729     }
1730     public void setPersonName(String personName)
1731     {
1732         this.personName=personName;
1733     }
1734     public String getPersonName()
1735     {
1736         return personName;
1737     }
1738   -----
1739   License.java
1740  import java.util.*;
1741  import javax.persistence.*;
1742  import org.hibernate.annotations.GenericGenerator;
1743  import org.hibernate.annotations.Parameter;
1744  @Entity
1745  @Table(name="License")
1746  public class License
1747  {
1748     @GenericGenerator(name="gen1",strategy="foreign",
1749                         parameters=@Parameter(name="property",value="person"))
1750     @Id
1751     @GeneratedValue(generator="gen1")
1752     @Column(name="lic_id")
1753     private int licenseld;
1754     @Column(name="idate")
1755     private Date issuedDate;
1756     @Column(name="edate")
1757     private Date expireDate;
1758     @OneToOne(targetEntity=Person.class,cascade=CascadeType.ALL)
1759     @PrimaryKeyJoinColumn
1760     private Person person;
1761     public void setLicenseld(int licenseld)
1762     {
1763         this.licenseld=licenseld;
1764     }
1765     public int getLicenseld()
1766     {
1767         return licenseld;
1768     }
```



```
1769     public void setIssuedDate(Date issuedDate)
1770     {
1771         this.issuedDate=issuedDate;
1772     }
1773     public Date getIssuedDate()
1774     {
1775         return issuedDate;
1776     }
1777     public void setExpireDate(Date expireDate)
1778     {
1779         this.expireDate=expireDate;
1780     }
1781     public Date getExpireDate()
1782     {
1783         return expireDate;
1784     }
1785     public void setPerson(Person person)
1786     {
1787         this.person=person;
1788     }
1789     public Person getPerson()
1790     {
1791         return person;
1792     }
1793 }
-----hibernate.cfg.xml-----
1794 <!DOCTYPE hibernate-configuration PUBLIC
1795   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
1796   "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
1797 <!-- hibernate.cfg.xml -->
1798 <hibernate-configuration>
1799   <session-factory>
1800     <!-- connection properties -->
1801     <property name="connection.driver_class">oracle.jdbc.OracleDriver</property>
1802     <property name="connection.url">jdbc:oracle:thin:@localhost:1521:satya</property>
1803     <property name="connection.username">scott</property>
1804     <property name="connection.password">tiger</property>
1805     <!-- hibernate properties -->
1806     <property name="dialect">org.hibernate.dialect.OracleDialect</property>
1807     <property name="show_sql">true</property>
1808     <property name="hbm2ddl.auto">update</property>
1809     <!-- mapping files -->
1810     <mapping class="Person"/>
1811     <mapping class="License"/>
1812   </session-factory>
1813 </hibernate-configuration>
1814 -----InsertClient.java-----
1815 import java.util.*;
1816 import org.hibernate.*;
1817 import org.hibernate.cfg.*;
1818 class InsertClient
1819 {
1820     public static void main(String[] args)
1821     {
1822         SessionFactory factory= new Configuration().configure().buildSessionFactory();
1823         Session session = factory.openSession();
1824         //Person object
1825         Person p = new Person();
1826         p.setPersonId(1101);
1827         p.setPersonName("ABC");
1828         //License object
1829         License l = new License();
1830         l.setIssuedDate(new Date());
1831         l.setExpireDate(new Date());
1832         l.setPerson(p);
1833         Transaction tx = session.beginTransaction();
1834         session.save(l);
1835         tx.commit();
1836     }
```



```
1837     session.close();
1838     factory.close();
1839   }
1840 }
=====
1842 Application-28(Component Test using Annotations)
1843 -----Person.java-----
1844 //Person.java
1845 import java.util.*;
1846 import javax.persistence.*;
1847 @Entity
1848 @Table(name="persons")
1849 public class Person
1850 {
1851     @Id
1852     @Column(name="pid")
1853     private int personId;
1854     @Embedded
1855     private PName pname;
1856     @Column(name="dob")
1857     private Date dob;
1858     public void setPersonId(int personId)
1859     {
1860         this.personId=personId;
1861     }
1862     public int getPersonId()
1863     {
1864         return personId;
1865     }
1866     public void setPname(PName pname)
1867     {
1868         this.pname=pname;
1869     }
1870     public PName getPname()
1871     {
1872         return pname;
1873     }
1874     public void setDob(Date dob)
1875     {
1876         this.dob=dob;
1877     }
1878     public Date getDob()
1879     {
1880         return dob;
1881     }
1882 };
1883 -----PName.java-----
1884 //PName.java
1885 import javax.persistence.*;
1886 @Embeddable
1887 public class PName
1888 {
1889     @Column(name="i",length=1)
1890     private char initial;
1891     @Column(name="fname",length=10)
1892     private String firstName;
1893     @Column(name="lname",length=10)
1894     private String lastName;
1895     public void setInitial(char initial)
1896     {
1897         this.initial=initial;
1898     }
1899     public char getInitial()
1900     {
1901         return initial;
1902     }
1903     public void setFirstName(String firstName)
1904     {
```



```

1905     this.firstName=firstName;
1906   }
1907   public String getFirstName()
1908   {
1909     return firstName;
1910   }
1911   public void setLastName(String lastName)
1912   {
1913     this.lastName=lastName;
1914   }
1915   public String getLastName()
1916   {
1917     return lastName;
1918   }
1919 }

1920 -----hibernate.cfg.xml-----
1921 <!DOCTYPE hibernate-configuration PUBLIC
1922   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
1923   "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
1924 <!-- hibernate.cfg.xml -->
1925 <hibernate-configuration>
1926   <session-factory>
1927     <!-- connection properties -->
1928     <property name="connection.driver_class">oracle.jdbc.OracleDriver</property>
1929     <property name="connection.url">jdbc:oracle:thin:@localhost:1521:satya</property>
1930     <property name="connection.username">scott</property>
1931     <property name="connection.password">tiger</property>
1932     <!-- hibernate properties -->
1933     <property name="dialect">org.hibernate.dialect.OracleDialect</property>
1934     <property name="show_sql">true</property>
1935     <property name="hbm2ddl.auto">update</property>
1936     <!-- mapping files -->
1937     <mapping class="Person"/>
1938     <mapping class="PName"/>
1939   </session-factory>
1940 </hibernate-configuration>
1941 -----Client.java-----
1942 import java.util.*;
1943 import org.hibernate.*;
1944 import org.hibernate.cfg.*;
1945 class Client
1946 {
1947   public static void main(String[ ] args)
1948   {
1949     SessionFactory factory = new Configuration().configure().buildSessionFactory();
1950     Session session = factory.openSession();
1951     PName p = new PName();
1952     p.setInitial('S');
1953     p.setFirstName("Abc");
1954     p.setLastName("Xyz");
1955     Person pr = new Person();
1956     pr.setPersonId(111);
1957     pr.setDob(new Date());
1958     //assign PName object to Person
1959     pr.setPname(p);
1960     Transaction tx = session.beginTransaction();
1961     session.save(pr);
1962     tx.commit();
1963     session.close();
1964     factory.close();
1965   }
1966 }
1967 ==
1968 Application-29(Easy Hibernate caching)
1969 -----Employee.java-----
1970 //Employee.java (POJO)
1971 public class Employee
1972 {

```



```

1973 private int employeeId;
1974 private String employeeName;
1975 private int employeeSal;
1976 private int deptNumber;
1977 public int getDeptNumber() {
1978     return deptNumber;
1979 }
1980 public void setDeptNumber(int deptNumber) {
1981     this.deptNumber = deptNumber;
1982 }
1983 public String getEmployeeName() {
1984     return employeeName;
1985 }
1986 public void setEmployeeName(String employeeName) {
1987     this.employeeName = employeeName;
1988 }
1989 public int getEmployeeId() {
1990     return employeeId;
1991 }
1992 public void setEmployeeId(int employeeId) {
1993     this.employeeId = employeeId;
1994 }
1995 public int getEmployeeSal() {
1996     return employeeSal;
1997 }
1998 public void setEmployeeSal(int employeeSal) {
1999     this.employeeSal = employeeSal;
2000 }
2001 }

-----employee.hbm.xml-----
2002 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
2003   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
2004 <!-- employee.hbm.xml -->
2005 <hibernate-mapping>
2006   <class name="Employee" table="employee">
2007     <cache usage="read-only"/>
2008     <id name="employeeId" column="empno"/>
2009     <property name="employeeName" column="ename"/>
2010     <property name="employeeSal" column="sal"/>
2011     <property name="deptNumber" column="deptno"/>
2012   </class>
2013 </hibernate-mapping>
2014 -----hibernate.cfg.xml-----
2015 <!DOCTYPE hibernate-configuration PUBLIC
2016   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
2017   "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
2018 <!-- hibernate.cfg.xml -->
2019 <hibernate-configuration>
2020   <session-factory>
2021     <!-- connection properties -->
2022     <property name="connection.driver_class">oracle.jdbc.OracleDriver</property>
2023     <property name="connection.url">jdbc:oracle:thin:@localhost:1521:satya</property>
2024     <property name="connection.username">scott</property>
2025     <property name="connection.password">tiger</property>
2026     <!-- hibernate properties -->
2027     <property name="dialect">org.hibernate.dialect.OracleDialect</property>
2028     <property name="show_sql">true</property>
2029     <property name="hbm2ddl.auto">update</property>
2030     <property name="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</property>
2031     <!-- mapping files --> <property name="hibernate.Usr_second-level-cache">true</property>
2032     <mapping resource="employee.hbm.xml"/> B  
true C (PROPRY)
2033   </session-factory>
2034 </hibernate-configuration>
2035 -----ehcache.xml-----
2036 <!-- ehcache.xml -->
2037 <ehcache>
2038   <defaultCache maxElementsInMemory="100"
2039     eternal="false" → eternal= false means we are setting some time  
for the objects in cache.
2040   

```

if we don't set any time limit then we should make eternal="true"



```
2041             timeToIdleSeconds="120"
2042             timeToLiveSeconds="200"/>
2043 <cache name="Employee"
2044     maxElementsInMemory="10"
2045     eternal="false"
2046         timeToIdleSeconds="8"
2047         timeToLiveSeconds="300"
2048     />
2049 </ehcache>
2050 =====client.java=====
2051 //client.java
2052 import org.hibernate.*;
2053 import org.hibernate.cfg.*;
2054 public class client
2055 {
2056     public static void main(String args[])
2057     {
2058         SessionFactory factory=new Configuration().configure().buildSessionFactory();
2059
2060         Session ses1 = factory.openSession();
2061         Session ses2 = factory.openSession();
2062         Session ses3 =factory.openSession();
2063
2064         Employee e1=(Employee)ses1.get(Employee.class,7876);
2065         System.out.println(e1.getEmployeeName());
2066         ses1.clear();
2067         System.out.println("session1 cleared");
2068         e1=(Employee)ses1.get(Employee.class,7876);
2069         System.out.println(e1.getEmployeeName());
2070         System.out.println("=====");
2071         try
2072         {
2073             Thread.sleep(6000);
2074         }
2075         catch(Exception e)
2076         {
2077
2078         Employee e2=(Employee)ses2.get(Employee.class,7876);
2079         System.out.println(e2.getEmployeeName());
2080         System.out.println("=====");
2081
2082         try
2083         {
2084             Thread.sleep(6000);
2085         }
2086         catch(Exception e)
2087         {
2088
2089         Employee e3=(Employee)ses3.get(Employee.class,7876);
2090         System.out.println(e3.getEmployeeName());
2091         ses1.close();
2092         ses2.close();
2093         ses3.close();
2094         factory.close();
2095     }
2096 }
2097 =====
```

I would try to update our site [JavaEra.com](http://JavaEra.com) everyday with various interesting facts, scenarios and interview questions. Keep visiting regularly.....

**Thanks and I wish all the readers all the best in the interviews.**

**www.JavaEra.com**

A Perfect Place for All **Java Resources**