HIBERNET NOTES

BY

SHEKHAR SIR

SATYATECHNOLOGIES

SRI RAGHAVENDRA XEROX

Software Languages Material Available

Beside Bangalore Ayyangar Bakery, Opp. C DAC, Ameerpet, Hyderabad.

Cell: 9951596199

sek

2/7/15

90/=

1) ORM (Object Relational Mapping) tools

Why ORM Tools are introduced?

⇒ A Typical real time JEE application contains 3 layers.

1) Presentation layer
2) Business layer (or) service layer
3) Persistence layer (or) DAO layer

⇒ In real time projects, data will be exchanged between the layers.

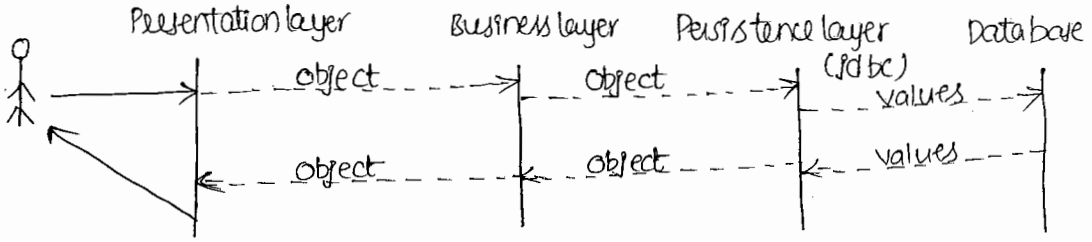
⇒ If the data is exchanged in the form of values then no. of trips b/w one layer to another layer will be increased and the performance of an application will be decreased.

⇒ To solve the above problem, data between the layers will be transfered in the form of objects.

⇒ In persistence layer, if jdbc technology is used it will transfer data in the form of values from application to data base (or) from data base to application.

⇒ A developer is responsible to transfer the data from object format to values format for inserting and values format to object while retreiving the data from database.

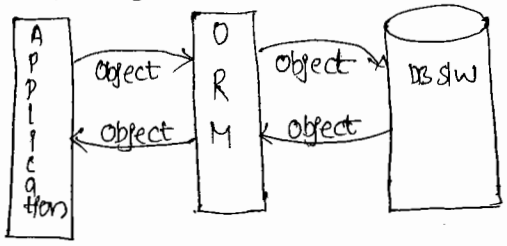
⇒ This conversion increases the burden on the developers.



⇒ In order to reduce the burden on developers, third party vendors started providing ORM tools

⇒ ORM tool directly transfers object from application to the database and reads object from database to application.

⇒ It means a developer is no need to transfer data from objects to values and values to objects explicitly.



03/7/15

More drawbacks of JDBC

1) In JDBC application, we create sql queries for database operations.

⇒ sql queries are database dependent so the persistence logic of jdbc is a database dependent.

2) jdbc exceptions are checked exceptions.

⇒ In a project, where ever we write the jdbc code, we need to put the code in a try & catch blocks. so jdbc increase the burden on the programmer.

3) In jdbc we use Resultset object (rs) to store the data selected from database. It is not transferable over the network.

⇒ We have an alternate called "Rowset" but all jdbc drivers doesnot support Rowset objects.

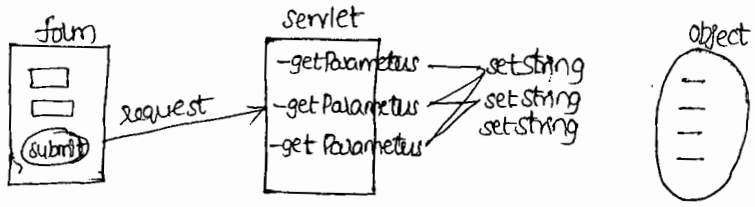
4) If the database table structure is modified after developing the jdbc project then that project does not execute. we need to modify the queries according to the new structure of the table in the entire project. It increase the burden on programmer and also it is a time consuming process.

* As a solution for the above drawbacks of jdbc technology, ORM tools are introduced in the market by the vendor.

Some list of ORM tools

1) Top link from oracle
2) MyBatis (iBatis) from Apache
3) JDO (Java Data Object) from Adobe
4) Hibernate from jboss
5) ORMLite
6) OpenJPA
7) Athena
8) OJB (Object Java Bean) etc..

Before framework



06/7/15

What really a framework ?

⇒ Initially when java was introduced, it has only JAVA Api to develop stand-alone applications.

⇒ Later a group of Api's are released under the name J2EE for developing distributed applications in JAVA.

⇒ Industry developers are faced some problems, while creating projects using java and J2EE api's.

⇒ J2EE is largeset of api's with many classes & interface. so it has become burden on developers to remember the api's.

⇒ A developer faced problems while integrating multiple api's in a project. A developer is need to write a lot of boiler-plate (repetative code).

⇒ To overcome the above problems, frameworks are started by the thired party vendors.

⇒ A framework provides a framework api, which is an abstraction on top of the java & J2EE api's.

⇒ A framework is not a new technology. It is an abstraction layer on top of existing technology.

⇒ With frameworks, we have following benefits.

1) A developer burden will be reduced
2) A project can be delivered to the clients fastly.
3) A project can be maintained easily.

Types of frameworks

1) Invasive / Intrusive framework

⇒ A developer has to extend a class from framework provided superclass (or) implement a class from framework provided interface.

Ex: struts and JSF

2) Non-invasive / non intrusive framework

⇒ Here developer does not extend a super class (or) to implement an interface of the framework.

Ex: Struts and hibernate and spring are non-invasive framework.

Hibernate

⇒ Hibernate is an open source, non-invasive framework from JBoss.

⇒ Hibernate is a middle layer between java and database. It transfers the data in the form of objects.

⇒ Hibernate can be used in any type of java projects, it means hibernate executes with out a server or even with in a server also.

swing api (standalone application) + Hibernate ⟶ Database → With out a server

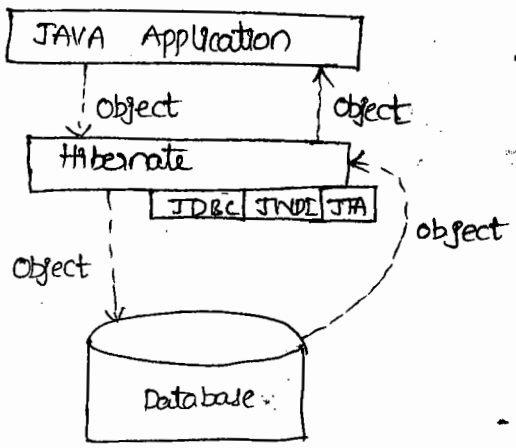
servlet api + Hibernate ⟶ Database → With in a server

⇒ Hibernate is an abstraction layer on top of following 3 technologies.

(1) jdbc (major technology)

(2) JNDI (Java Naming and Directory Interface)

(3) JTA (Java Transaction API)



04/07/15

Benefits of Hibernate

1) HQL (Hibernate Query Language):

⇒ Hibernate has introduced its own query language with name HQL.

⇒ While working with JDBC technology, SQL queries are used in a Application. A problem faced is, the application persistence logic is database dependent.

⇒ To make persistence logic as database independent, hibernate has introduced HQL.

⇒ HQL does not provide any burden on the developers, because it looks like SQL only.

⇒ A difference between SQL & HQL, SQL commands are database dependent but HQL commands are database independent.

Ex: SQL : select empno, sal from emp;

HQL : select e.employeeNumber, e.employeeSalary from Employee e;

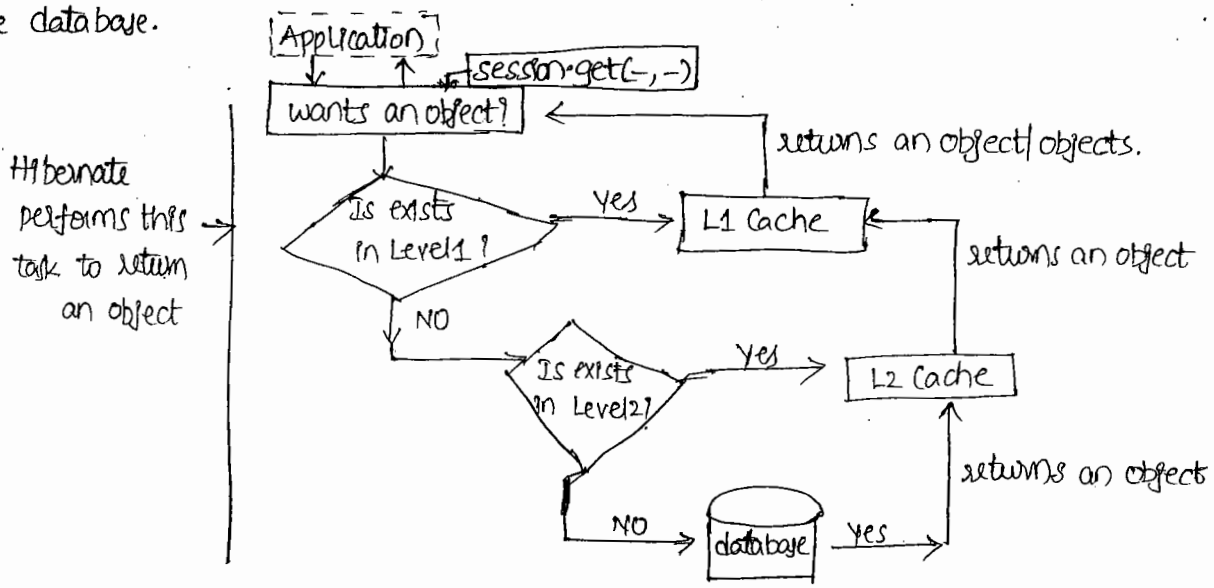
2) Caching

⇒ It is a most important feature (or) benfit of hibernate.

⇒ A cache reduces the no.of round trips between an application and a database, it improves performance of an application.

⇒ Hibernate maintains cache at 2 levels, so hibernate gives better performance.

⇒ When an application wants an object from database, then hibernate looks for that object at level1, if not then looks for it in level2 and if not then only it goes to the database.



08/07/16

3) Lazy loading

⇒ Lazy loading is also an important feature which improves the performance by reducing network round trips between java application and database.

⇒ In Lazy loading, hibernate does not load an object from database immediately, an object will be loaded on demand.

⇒ If an object is immediately loaded then it is called "eagly loading". /"early Loading".

⇒ In Lazy loading, hibernate creates a proxy, returns proxy objects to the application.

⇒ When application is accessing the proxy object then internally data will be selected from database.

⇒ With Lazy loading feature, no of trips with the database are going to be reduced and the performance will be improved.

4) Criteria

⇒ We can read the same output from database by executing the different SQL commands.

Eg. 1) select * from emp.

2) select empno, ename, sal from emp.

* The above 2 different sql commands will return same output but 2nd queries gives better performance

⇒ While reading the data from database, tuned queries are important to improve the performance.

⇒ As a java developer, creating tuned queries will increase burden on the developer.

⇒ To decrease the burden, hibernate as provided criteria api, it internally creates tuned queries and executes them on the database.

⇒ So a developers is no need to prepare tuned queries explicitly.

5) Locking

⇒ To restrict multiple threads for updating same record of the database, Hibernate introduced locking facility.

Hibernate has provided 2 types of locking.

1) Optimistic Locking
2) Pessimistic Locking

⇒ With the 2types of locking we can make the database transactions as following ACID principles. (Atomicity, Consistency, Isolation, Durability)

" A single logical operation on the data is called "transaction".

6) Un-checked exception

⇒ In hibernate, all exceptions are unchecked type, so there is no need add try and catch blocks for the hibernate logic.

⇒ Hibernate internally uses jdbc technology and jdbc exceptions are checked type, so hibernate has a translator internally, which converts checked expection of jdbc to the un-checked exception of hibernate.

Files required to write a hibernate application are

1) POJO class
2) Mapping file
3) Configuration file
4) Client Application

Java Bean

⇒ The java class i.e., developed with some standards is called Java Bean. It is helper class in application development to represent data in the form of objects & to send the over the network. If the java class satisfies the following standards then it is called Java Bean. ie,

a) Class must be public class & must implement java.io.Serializable(I)
b) Must have private properties (member variables)
c) Every property must have one setter method & one getter method
d) Must have explicitly placed or implicitly generated 0-param constructor (By java compiler)

NOTE Setter methods are useful to set data bean properties & getter methods are useful to read data from bean properties.

Example

```
package com.s.h.bean;
Public class StudentBean implements java.io.Serializable
{
    //Bean Properties
     private int no;
     Private String name;
     //0-param constructor
     public StudentBean(){}
        }
    //setters and getters--
}
```

Q Difference b/w POJO class & JAVA BEAN?

A Pojo class need not to have accessor methods (getter and setter) & need not to fulfill other standards of Java Bean. Java Bean must have accessor methods & must satisfy other standards. Eventhough it is satisfied java bean standards its look like POJO class.

CONCLUSION Every Java Bean is a POJO class. But every class need not be Java Bean.

Domain Class

⇒ The Java class whose object can represents DB Table record values (colum values) is called Domain class | Business Object Class | Entity class | Model class.

⇒ This class should have properties to hold the column values of DB Table record.

Domain class: a) May or may not be a public class
b) May or may not implements Serializable (I)
c) Properties (member vaiables) can be private or public or protected or default variables.
d) Must have explicit or implicit 0-param Constructor.
e) Must have setter or getter methods for properties.

⇒ we can take java bean class as Domain Class
⇒ we can say every Domain Class is POJO class.
⇒ In HB the class whose object represents DB table records in ORMapping will be taken as Domain class.
⇒ Domain class will be mapped with Table, its properties will be mapped with DB table's columns & its objects represent db table records having synchronization.

DBTable

~~Serializable~~

student

- sno (number)
- sname (varchar2)

Domain class

```
public class implements Serializable
{
    private int no;
    private String name;
    public Student()
    {
    }
    //setters and getters
    ---
}
```

09/7/15

1) POJO class

⇒ POJO is a standard, given for indicating a normal java class.

⇒ A JAVA class, which does not exceed the boundary of java api is called a "POJOclass"

⇒ In other words, if a class is compiled by java compiler with out taking the support of any jar file then that java class can be called as a "POJO class."

Ex1
```
class A                    // A is a POJO class.
{
  void m1()
  {
  }
}
```

Ex2
```
public class A implements Runnable    // A is a POJO class.
{
  public void run()
  {
  }
}
```

Ex3
```
public class A extends HTTPServlet    // A is not a POJO class.
{
  public void doGet(request,response){
  }
}
```

Ex4
```
class A implements Remote    // A is a POJO class
{
}
class B extends A            // B is a POJO class.
{
}
```

JAVA BEAN

⇒ A JAVA class can be called as a JAVA Bean, if the class satisfies the following rules.

1) class must be public
2) class must contain a public default constructor
3) Each private variable must contain a public setter/getter/both methods
4) A class can atmost implement java.io.Serializable (interface)

Note Every Java Bean is a POJO class, but every POJO class is not a Java Bean.

Ex1
```
public class A                    // A is a Java Bean
{                                 // A is a POJO class.
  private int x;
  public void setX(int x){
     this.x=x;
   }
}
```

Ex2
```
public class A                    // A is a POJO class
{                                 // A is not a Java Bean
  private int x;
  public A(int x){
    this.x=x;
   }
 public int getX(){
  return x;
   }
}
```

Ex3:
```
public class A implements Serializable, Clonable    // A is a POJO class
{                                                    // A is not JavaBean
  public Object clone() throws CloneNotSupportedException
     {---
     }
}
```

10/7/15

(8) Mapping file

⇒ A Java Application can have multiple classes and database can have multiple tables.

⇒ Hibernate knows how to store an object and how to read it, but it does not know which java class object need to be persisted (stored) in which table of database.

⇒ It's programmer responsibility to tell the hibernate that which class object needs to be stored in which table. to pass this information to hibernate as a programmer we need to construct the mapping files.

⇒ A mapping file contains mappings b/w POJO class to table and POJO class variables to columns of the table.

⇒ for a mapping file, because it is an xml file, extension of the file should be .xml. But hibernate is recomanded to use extension as .hbm.xml, to recognise a mapping file easily.

⇒ A sample mapping file, for mapping employee class to emp table will be like the following,

employee.hbm.xml

```
<hibernate-mapping>
    <class name="Employee" table="emp">
        <id name="employeeNumber" column="empno" />
        <property name="employeeName" column="ename" />
        <property name="employeeSalary" column="sal" />
        <property name="deptNumber" column="deptno" />
    </class>
</hibernate-mapping>
```

(Java Bean class name) (table name in DB) (primary key)

Important statements

⇒ we can map multiple classes in a single hbm.xml file.

⇒ under <class> tag, it is mandatory to write <id> tag (or) <composite-id> tag.

⇒ <id> tag is to map id variable of a class to primary key column of a table.

⇒ <property> tag is to map non-id variable of a class to non-primary key column of a table

⇒ If there are no non-primary key variables in a class then we can avoid <property> tag, we cannot avoid <id> tag.

⇒ If, table name of database is same as class name then we can avoid table attribute

in <class> tag.

⇒ If column name is same as variable name then we can avoid column attribute in <id> and <property> tags.

Q How many mapping files are created in a project?

A) Generally, in real-time projects, one mapping file will be created for each module, in which there is need to access the database.

3) Configuration file:

⇒ It is another xml file in a hibernate application and it consists the following 3 sections,

1) connection properties
2) hibernate properties
3) mapping resources

⇒ In hibernate, it internally opens connection with the database and also it closes automatically.

⇒ As a developer, we need to provide connection properties to hibernate through configuration file.

⇒ Apart from primary features added by hibernate for an application, if any additional features are required then we need to attach hibernate properties in configuration file.

⇒ Into an application mapping files are need to be loaded along with configuration file only. so we need to attach location of mapping files in configuration file.

⇒ A configuration file is an xml file, so extension required is .xml, but hibernate has recomanded .cfg.xml as extension file for configuration file.

⇒ Connection properties and hibernate properties can be shuffled, but mapping resources must be at bottom of the xml file only.

⇒ ****
In a hibernate project, we need a seperate configuration file for each database.
It means the no. of configuration files in a projects depends on the no. of databases.

11/7/15 hibernate.cfg.xml

```
<hibernate-configuration>
    <session-factory>
     <!-- connection properties -->
  <property name="connection.driver_class">oracle.jdbc.oracledriver</property>
  <property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
  <property name="connection.username">system</property>
  <property name="connection.password">tiger</property>
compulsory <!-- hibernate properties -->
  <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>
     <!-- Echo all SQL executed to stdout -->
  <property name="hibernate.show-sql">true</property>
     <!-- Drop and re-create the database schema on the start-up -->
     <!-- mapping resources --> <property name="hbm2ddl.auto">update</property>

  <mapping resources="employee.hbm.xml" />

  </session-factory>
</hibernate-configuration>
```

(4) Client Application:

Steps(1) ⇒ Writing a java program to perform any persistent operations with support of hibernate is loading a configuration file.

⇒ Hibernate api has provided configure() method in Configuration class.

⇒ By creating an object of Configuration class, hibernate environment is started in a java application.

⇒ When configuration file loaded then along with that file, mapping files are also loaded.

⇒ Internally an xml file parser (DOM parser) will check the well-formed-ness and valid-ness of xml files.

⇒ If valid, then data will be loaded from xml-files into different variables of Configuration class.

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

STEP(2):

⇒ In this step, we need to get all the data of all xml files into a single object, Sessionfactory object.

⇒ Sessionfactory is an interface of hibernate api.

⇒ We can get a Sessionfactory object by calling buildSessionFactory() method of configuration class.

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

STEP(3):

⇒ We cannot directly work with the SessionFactory object. It is a factory for producing Session objects.

⇒ A Session object is a main object through which we can perform CURD operations

⇒ When a session object is created then internally that session object opens a connection with the database.

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

⇒ Session is an interface of hibernate api.

STEP(4):

⇒ In hibernate, to perform non-select operations on DB, a transaction must be started.

* For a select operation, a transaction is optional.

⇒ In this step, we begin transactions and commit after completion of transactions.

⇒ If any exception, hibernate automatically rollback the transactions.

```
Transaction tx = session.beginTransaction();
//operations
tx.commit();
```

STEP(5):

⇒ close a session.

⇒ When a session is closed then internally a connection will also be closed.

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

STEP(6)

⇒ Finally we need to close the sessionfactory, when the application before going to shutdown.

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

13/7/15

Hibernate software

⇒ A framework is an abstraction layer on top of existing technology.

⇒ A framework software will be released as a set of Jars.

⇒ When we download a zip file will be downloaded and when it is extracted the software of what framework is installed.

⇒ In hibernate, a version used in industry is 3.6.10 and currently released version is 4.3.10

⇒ We can download hibernate software from http://sourceforge.net/projects/hibernate/files/hibernate3/3.6.10.final/

Jars dependency: When we are creating our applications, we take the suppoit of a classes that is in a jar file and that class takes the suppoit of a class that is in another jar file. so we say that our jar is depending on another jar.

for Ex: MyProgram.jar

```
public class MyProgram {
    public static void main (String args[]) {
        X x = new X();
    }
}
```

a.jar

```
public class X
{
    Y y = new Y();
}
```

b.jar

```
public class Y {
---
}
```

C:\> javac MyProgram.java (error)

(error generated because X class is not a parsed in java api)

C:\> set classpath E:\a.jar;.;

C:\> javac MyProgram.java

(compiled successfully because X class is existed in a.jar)

C:\> java MyProgram

(Exception, because X class depends on Y class of b.jar)

C:\> set classpath E:\a.jar;E:\b.jar;.;

C:\> java MyProgram

(Executed successfully)

⇒ To MyProgram class, main jar is a.jar and dependency jar is b.jar.

⇒ To run hibernate applications we need to add the following list of jars to the classpath variable.

1) hibernate3.jar (D:\hibernate-distribution-3.6.10.Final)
2) antlr-2.7.6.jar (D:\hibernate-distribution-3.6.10.Final\lib\required)
3) commons-collections-3.1.jar (" " " ")
4) dom4j-1.6.1.jar (" " " ")
5) javassist-3.12.0.jar (" " " ")
6) jta-1.1.jar (" " ")
7) slf4j-api-1.6.1.jar (" " " ")
8) hibernate-jpa-2.1-Final.jar (D:\hibernate-distribution-3.6.10.Final\lib\jpa)
9) driver jar file.

First Application

Folder1
↳ Product.java (javabean class)
Product.hbm.xml (mapping file)
hibernate.cfg.xml (configuration file)
Insert.java (client application)

Product.java.

```
public class Product {
    private int productId;
    private string productName;
    private double price;
  //setter() and getter() methods
    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;
    }
}
```

Product.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC -//Hibernate/Hibernate Mapping DTD 3.0//>
<hibernate-mapping>
 <class name="Product" table="Product">
    <id name="productId" column="pid" />
    <property name="productName" column="pname" />
    <property name="price" column="price" />
 </class>
</hibernate-mapping>
```

hibernate.cfg.xml

```
<hibernate-configuration>
 <session-factory>
 <property name="Connection.driver_class"> oracle.jdbc.driver.OracleDriver </property>
 <property name="Connection.url"> jdbc:oracle:thin:@localhost:1521:xe </property>
 <property name="Connection.username"> system </property>
 <property name="Connection.password"> tiger </property>
 <property name="hibernate.dialect"> org.hibernate.dialect.OracleDialect </property>
 <property name="hibernate.show_sql"> true </property>
 <mapping resource="Product.hbm.xml" />
 </session-factory>
</hibernate-configuration>
```

15/7/15

Insert.java

```
import org.hibernate.cfg.Configuration;
import org.hibernate.SessionFactory;
import org.hibernate.Session;
import org.hibernate.Transaction;
class Insert {
public static void main (String args[]){
```

step1

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

step2

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

step3

```
Session session = factory.openSession();
// create POJO class object
    Product p = new Product();
        p.setProductId(101);
        p.setProductName ("iphone");
        p.setPrice (6000);
```

step4

```
Transaction tx = session.beginTransaction();
        session.save(p);
        tx.commit();
    S.O.P ("object is saved to database");
```

step5

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

step6

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

Compilation : javac ~~Product.java~~ Product.java

Before compiling Insert.java, Add all the required jars of hibernate and jdbc drivel jar file (ojdbc14.jar) to the classpath variable.

D: javac Insert.java

D: java Insert

O/P

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

object is saved on database.

⇒ To verify whether object is saved in database (or) not. Type the following command into SQL.

SQL > select * from Product;

PID	PNAME	PRICE
101	iphone	6000

hibernate.hbm2ddl.auto → Property

1) validate (default)
2) create
3) update
4) create-drop

⇒ If the value is "validate" then hibernate only verifies the table and columns are exists in the database (or) not. If exists then operation is executed and if not then throws an exception.

⇒ If the value is "create" then hibernate first drops the existing table then creates a new table and then executes the operation on that table.

<property name="hibernate.hbm2ddl.auto">create</property>

⇒ If the value is "update" then hibernate checks whether a table exists in database or not. If exists then executes operation on the same table if not exist then it will create a new table and executes operation on that table.

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

create-drop

⇒ If the value is "create-drop" then hibernate creates a new table, executes operations and finally drops that table. This value will be used by performing unit testing.

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

16/7/15

Student.java

```
public class student
{
    void m1()
    {
    }
}
```

compiled

Student.class

```
public class student{
    public static Class class; → // property
    public student(){ } → // constructor
    void m1(){
    }
}
```

(class name → Class, keyword → class, property → class)

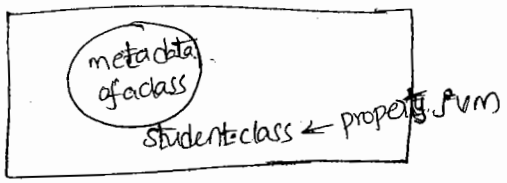
compiled

Class → class name
class → keyword
class → property.

⇒ We want to access the static class variable.

```
Class c = Student.class;
```

↑ class object containing meta data of student class.



metadata of the class contains

- → modifiers
- → public constructor
- → class name of a class
- → variables of a class
- → Type of class
- → methods of a class.
- → super class of the class.
- → Interfaces of the class.

Reading an object from database.

⇒ Hibernate has given following 2 methods to read an object from database.

1) load()
2) get()

⇒ To load an object from database, we need to pass a POJO class whose "object" we want to load and "id" of that object.

⇒ We can pass either name of the POJO class as a string (or) Class object of a POJO class.

Syntax
```
load(String classname, Serializable id)  // load("Product", 101);
load(Class c, Serializable id)           // load(Product.class, 101);
```

get(String classname, Serializable id) // get("Product", 101);

get(Class c, Serializable id) // get(Product.class, 101);

⇒ When we compile a class in java, internally compiler adds class property(variable) for each class. It is a static property.

⇒ At loading time, JVM creates a class object for the loaded class and stores it in "class" property.

⇒ Class object stores metadata of a class. Metadata of a class means

a) Modifiers of a class
b) Name of the class
c) Super class name
d) Interfaces implemented by the class
e) fields of the class
f) Constructor
g) Methods of the class

⇒ We can read Class object of a class by calling class property.

Eg. 1) Class P = Product.class;
2) Class S = Student.class;

Note

A
↑
B

B extends A

A a = new B(); ← we can store sub class object in super class reference (sub class object reference can be of type super class also)

B b = new A(); ✗ → This is wrong in java.
↳ superclass object is not of type subclass.

Early loading : Automatically database picks the data into to application. Here trips will waste if not required. It will decrease the performance.

Lazy loading : If required only it will get the data from database to application. It will improve the performance.

How load() method works!

⇒ When we call load() method it does not goes to the database immediately.

⇒ First load() method assumes that object exists in database, then creates a proxy class and then object.

⇒ load() method will set "id" to the proxy object and then returns that proxy object.

⇒ When we call a non id Property method then hibernate goes to database then verifies the id, if exists then reads its data and it will set the data to the proxy object. This type of loading is called "lazy loading".

⇒ If the id does not exists in the database then this load() method throws ObjectNotFoundException.

How get() method works?

⇒ get() method will immediately goes to database, verifies id and if exists then reads that object from database.

⇒ get() method does not create any proxy data, so it is called "eagly loading".

⇒ If the "id" does not exists in database, it returns null values. It does not throw any exception.

17/7/15

// select.java

```
import org.hibernate.cfg.Configuration;
import org.hibernate.SessionFactory;
import org.hibernate.Session;
public class select{
    public static void main(String[] args){
        Configuration conf = new Configuration();
            conf.configure("hibernate.cfg.xml");
        SessionFactory factory = conf.buildSessionFactory();
        Session session = factory.opensession();
        Object o = session.get(Product.class, 101);
        Product p = (Product)o;
```

```
String str = p.getProductName();
S.O.P("pname = "+str);
session.close();
factory.close();
}
}
```

```
//for load() method
Object o = session.load(Product.class, 101);
Product p = (Product) o;
int x = p.getProductId();
S.O.P("pid = "+x);
```

```
String str = p.getProductName();
S.O.P("Pname = "+str);
```

It will return proxy class object having id 101 it won't goes to db. Now it goes to db, and returns product object with id (101).

Q What is the difference bt load() and get() methods?

⇒ load() method is used to Lazily load an object.
get() method is used to easily load an object

⇒ If the given "id" does not exists in database, load() method throws ObjectNotFoundException but get() method returns "null" value.

Q Why setter() and getter() methods are mandatory in a POJO class of hibernate

⇒ If there are no getter() methods then problem occurs while performing save() operation.

⇒ If there are no setter() methods then problem occurs in load() operation.

⇒ To execute all operation with out any problems, we must define both setter() and getter() methods for each variable of POJO class.

Q Why a default constructor is mandatory in a POJO class of hibernate.

⇒ While reading an object from database, internally hibernate creates an object of POJO class by calling default constructor.

⇒ If there is no default constructor in a class then its objects cannot loaded from database..

Q What is the need of a dialect class?

⇒ A dialect class generates SQL commands internally for hibernate based on the database.

⇒ Hibernate has already provided predefined dialect classes for almost each database.

⇒ We can find the list of the databases supported by hibernate and their respective dialects D:\hibernate-distribution-<version>\project\etc\hibernate.properties file.

20/7/15

Session Level Cache / Level1 Cache

⇒ Cache is a buffer allocated in RAM.

⇒ Caching is a mechanism which reduces database interactions and improves the performance of an application.

⇒ In hibernate a cache is managed by two objects 1) session object 2) sessionfactory object.

⇒ A cache managed by session object is called "Level1 cache (or) local cache".

⇒ A cache managed by factory object is called "Level2 cache (or) Global cache".

⇒ Level1 cache will be automatically created by hibernate, when a session is opened and it will be automatically closed when a session is closed.

⇒ As a programmer, we no need to add any special tags in a hbm file (or) in a cfg file to work with level1 cache.

⇒ This level1 cache is not in the hands of a programmer. It is in the hands of hibernate.

⇒ Each session will have its own cache. A session cannot read objects from another session cache.

⇒ As a programmer, we can delete either a specific object (or) all objects from cache, but not cache.

⇒ By using evict() method we can delete a particular object. If we want to delete all the objects from cache we can use clear() method.

⇒ When we call evict() method if the "id" of an object does not exists in the cache then evict() method throws an exception. Before evicting, we can check an object exists in cache (or) not with the help of contains() method.

Eg
```
if( session.contains(o) )
{                    ↑ object
  ↳ true
  session.evict(o);
}
```
↳ removes an object from cache.

⇒ A problem with cache is, if an object is loaded updated in database then the changes are not automatically reflected on cache.

⇒ It is a programmer responsibility to refresh a cache at a regular intervals, to get the changes from database to the cache.

For eg:

```
First select → Object o1 = session.get(Product.class, 101); // get from database first time
               Object o2 = session.get(Product.class, 101); // get from cache already id exists in cache
               session.evict(o1);                           // clears the object with id o1.
Second select ← Object o3 = session.get(Product.class, 101); // get from database again
New session third select ← Object o4 = session2.get(Product.class, 101); // get from database becoz new session
```

⇒ For the above code, 3 selects are generated by hibernate.

Updating an object

⇒ An object can be updated in 2 ways.

1) By without reading it from the database
2) By reading it from the database.

Approach(1)

⇒ If we want to update an object by without loading it from database. we must set each property before updating it.

⇒ If we donot set any property then it will take default value.

Approach(2)

⇒ We can only set the required properties, so comparatively approach 2 is recommanded then approach1.

Example Approach1

Ex:

```
Product p = new Product();
p.setProductId(101);
p.setProductName("iPhone");
p.setPrice(80000);
Transaction tx = session.beginTransaction();
session.update(p);
tx.commit();
```

Ex2 on Approach2

```
Object o = session.get(Product.class, 101);
Product p = (Product)o;
p.setPrice(8000);
Transaction tx = session.beginTransaction();
session.update(p);
tx.commit();
```

Deleting an object

⇒ An object can be deleted from database in 2 ways.

1) By with out reading it from database

2) By reading it from database.

Approach 1

⇒ To delete an object, we only need to set the "id" property.

Approach 2

⇒ Comparatively approach 2 is better than approach 1, before deleting we can check whether an object exists in database (or) not.

Ex1 on Approach1

```
Product p = new Product();
p.setProductId(101);
Transaction tx = session.beginTransaction();
session.delete(p);
tx.commit();
```

Ex2 on Approach2

```
Object o = session.get(Product.class, 101);
if(o!=null)
{
  session.delete(o);
}
else{
  S.O.P("sorry, object does not exists in database");
}
```

o contains (101) object details of Product POJO class.

~~Product p=~~

2/7/15

Alternative ways to configure in Hibernate

⇒ In hibernate configuration can be done in 3 ways.

1) By creating cfg.xml file
2) By creating a properties file
3) By directly setting in source code

⇒ In the above 3 ways, the best way to do configuration is, by creating cfg.xml file

Approach 1 (Problem)

⇒ We can directly set connection properties, hibernate properties and we can add the mapping resources in source code only.

for eg:

```
Configuration conf = new Configuration();
        conf.setProperty("connection.driver_class", "oracle.jdbc.driver.
        ------                                       OracleDriver");
        conf.setProperty("___", "___");
        -------
        conf.setProperty("hibernate.dialect", "org.hibernate.oracle10gdialect");
        ----
      conf.addResource("product.hbm.xml");
   SessionFactory factory = conf.buildSessionFactory();
```

Drawbacks

⇒ We have the following 2 drawbacks with directly writing the properties in source code.

1) Increases the source code.
2) We need to recompile, reload & then restart the server, if we want to change from one database to another database.

Approach 2 (problem)

* We can construct a properties file also like the following.

hibernate.properties

```
connection.driver_class = oracle.jdbc.driver.OracleDriver
connection.url = jdbc:oracle:thin:@localhost:1521:xe
connection.username = system
connection.password = tiger
hibernate.dialect = org.hibernate.dialect.Oracle10gDialect
hibernate.show_sql = true
hibernate.hbm2ddl.auto = update
```

⇒ In properties file, we can add connection and hibernate properties, but we cannot add mapping resources.

⇒ If the filename is hibernate.properties then it will be loaded automatically when configuration class object is created.

```
configuration conf = new Configuration();
conf.addResource("product.hbm.xml");
```

⇒ If the properties file is other than hibernate.properties, then that file will not be automatically loaded. We need to write some extra code to load that properties file.

For ex:

```
Configuration conf = new Configuration();
Properties props = new Properties();
FileInputStream fis = new FileInputStream("xyz.properties");
props.load(fis);
conf.setProperties(props);
conf.addResource("product.hbm.xml");
```

Approach3 (solution)

⇒ If it is cfg.xml file, we can write connection properties, hibernate properties & mapping resources also, so configuration xml (cfg.xml) is better approach than the remaining two approaches.

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

Generator Classes in hibernate

⇒ A generator class generates "id" for an object.

⇒ Hibernate calls a generator class, before going to insert an object into a database.

⇒ A generator class return an "id" & hibernate will set that "id" to an object and then inserts that object to database.

⇒ There are multiple predefined generator classes are exists in hibernate. If existing generators are not enough then we can also create a custom generator class.

⇒ In hbm file, under <id> tag, we need to configure <generator> tag as a subtag, to tell the hibernate about which generator class it should call, before going to

insert an object to database.

⇒ For all the predefined generator classes, there are alias names. so in hbm file we need to configure the alias name.

① assigned:

⇒ assigned is an alias for Assigned class.

⇒ If we dont configure generator tag in hbm file under <id> tag then hibernate bydefault uses Assigned generator class.

⇒ Assigned class reads "id" set by the programmer and returns the same "id" to hibernate. Finally hibernate will insert object in database with the same "id" set by the programmer.

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

(or)

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

⇒ Assigned is a database independent generator class, it means this generator works for any database.

22/7/15

② increment:

⇒ increment is an alias for IncrementGenerator class.

⇒ IncrementGenerator class selects maximum of the primary key value of the table then increments it by 1 and then returns that value to the hibernate.

⇒ Hibernate will insert an object with that "id" to the database.

⇒ increment is a database independent generator.

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

increment = maxvalue(id) +1 in database as primary key value.

③ sequence:

⇒ sequence is an alias name for SequenceGenerator class.

⇒ SequenceGenerator class reads next value of a sequence from database and then returns that value as "id" to the hibernate.

⇒ Hibernate will insert an object with that id to the database.

⇒ A database can have many sequences. so we need to pass a sequence name as a parameter to the SequenceGenerator class.

⇒ If we donot pass a sequence name as a parameter, then SequenceGenerator will first creates its own sequence in database with name hibernate_sequence and then reads the next value of that sequence.

⇒ sequence is a database dependent generator. It works with oracle, but doesnot work in mysql.

```
<id name="productId" column="pid">
    <generator class="sequence">
        <param name="sequence">test_sequence </param>
    </generator>
</id>
```

⇒ SequenceGenerator class selects next value of test_sequence from database.

⇒ If test_sequence doesnot exists in database an exception will be thrown.

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

⇒ SequenceGenerator class creates its own sequence with name hibernate_sequence and returns nextvalue to the hibernate.

④ hilo:

⇒ hilo is an alias name for TableHiLoGenerator class.

⇒ TableHiLoGenerator class returns "id" as '1' for first time.

⇒ This Generator class will store a count in a table, which indicates how many times "id" generated by the generator.

⇒ By default this generator class creates a table with name "hibernate_unique_key" with column "next_hi" and updates the value of a column, everytime when "id" is generated.

⇒ From second time onwards, this generator class uses a formula,

`max_lo * next_hi + next_hi` to generate the "id" value.

max_lo → is a parameter and its default is "32767".

next_hi → is a column name which contains count value indicating how many times "id" generated by the generator.

⇒ We can change the default value of max_lo parameter with our own value, by configuring "max_lo" parameter in hbm file.

⇒ This hilo generator has 3 parameters,

① table → default : hibernate_unique_key
② column → default : next_hi
③ max_lo → default : 32767

⇒ We can change the default values of all the 3 parameters by configuring in hbm file.

```
<id name="productId" column="pid">
  <generator class="hilo">
    <param name="table">MyTable</param>
    <param name="column">col1</param>
    <param name="max_lo">10</param>
  </generator>
</id>
```

MyTable

col1
1

max_lo = 10

First value = 1
10*2 + 2 = 22 second value

⇒ Hilo is a database independent generator. It means, we can use hilo for any generator for any database.

⑤ Identity :

⇒ Identity is an alias name for "IdentityGenerator class"

⇒ IdentityGenerator calls auto increment algorithm of a database, reads the incremented value and returns that value as "id" to the hibernate.

⇒ autoincrement algorithm does not exists for all databases. For ex, it exists in mysql, but does not exists in oracle.

⇒ Identity is a database dependent generator.

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

⑥ native :

⇒ native generator acts as either sequence / identity / hilo.

⇒ Hibernate first calls SequenceGenerator, if not supported then it calls IdentityGenerator, If not supported then finally TableHiloGenerator is called.

⇒ native is a database independent generator.

⑦ foreign :

⇒ foreign is an alias name for "ForeignGenerator" class.

⇒ This ForeignGenerator is used only in one-to-one relationship.

⇒ In one-to-one relationship, there is a need to copy parent record primary key value to child record, for this foreignGenerator class is used.

⇒ ForeignGenerator class reads primary key value of parent record and returns it to the hibernate. Hibernate inserts child record with that primary key in database

⇒ foreign is a database independent generator.

24/7/15

⑧ uuid.hex : (Universal unique id)

⇒ uuid algorithm generates a string as a primary key value based on the following

1) IP address
2) JVM start-up time
3) current system time
4) counter value in JVM.

⇒ When primary key column is string type then we can use predefined generator classes either assigned or uuid.hex.

⇒ The remaining generator classes like increment, sequence, hilo etc.. can be used, only when a primary key column is an integer type.

⇒ It is not suggestable to use uuid.hex, because it generates a lengthy primary key value with 32 characters.

⇒ When a primary key column is a string type it is suggestable to create a custom id generator class.

CustomGenerator

⇒ Every generator class whether it is a predefined generator class(or) user-defined generator class, must implement org.hibernate.id.IdentifierGenerator(interface).

⇒ IdentifierGenerator(interface) has only one abstract method called generate(). We need to override this generate method and we need to define the generator logic in this method.

```
public class MyGenerator implements IdentifierGenerator {
    @Override
    public Serializable generate(SessionImplementor si, Object o)
    {
        // logic
    }
}
```

⇒ SessionImplementor is used to get a connection with database in the logic.

⇒ A second parameter object is a POJO class object, its non-primary key properties values can be used in the logic.

⇒ For a custom id generator class, we cannot provide alias name. So in hbm file, we need to directly configure the generator classname in <generator> tag.

Ex:

In the following example, we are creating the custom generator class for creating customer id's like c001, c002, etc.. c010, ... c099, c100, ... c999.

Folders

- Customer.java
- Customer.hbm.xml
- hibernate.cfg.xml
- MyGenerator.java
- Insert.java

customer.hbm.xml

```
<hibernate-mapping>
  <class name="customer" table="customer">
    <id name="customerId" column="custid" type="java.lang.string" length="10">
     <generator class="MyGenerator" />
     </id>
    <property name="customerName" column="custname" type="java.lang.string"
                                  length="10" />
    <property name="location" column="location" type="java.lang.string" length="10" />
    </class>
</hibernate-mapping>
```

hibernate.cfg.xml

```
<hibernate-configuration>
<session-factory>
<property name="Connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="Connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
<property name="connection.password">system</property>
<property name="hibernate.dialect">tiger</property>
<property name="hibernate.show_sql">true</property>
<property name="hibernate.dialect">org.hibernate.dialect.OracleDialect</property>
<mapping resource="customer.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

// Customer.java

```
public class Customer {

    private String customerId;
    private String customerName;
    private String location;

    public void setCustomerId(String customerId) {
        this.customerId = customerId;
    }

    public String getCustomerId() {
        return customerId;
    }

    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }

    public String getCustomerName() {
        return customerName;
    }

    public void setLocation(String location) {
        this.location = location;
    }

    public String getLocation() {
        return location;
    }

}
```

MyGenerator.java

```
Public class MyGenerator implements IdentifierGenerator {
  @override
  Public Serializable generate (SessionImplementor si, Object o) {
    String id = " ";
    try {
    Connection con = si.connection();
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("select new_sequence.nextval from dual");
      while(rs.next()){
    int i = rs.getInt(1);
        if(i<=9)
          id = "COO"+i;
        else if(i>9 && i<=99)
            id = "CO"+i;
        else
            id = "C"+i;
      }rs.close();
       stmt.close();
       con.close();
      }
    catch (Exception e) {
      s.o.p(e);
    }
    return id;
    } // method close
}// class close
```

Insert.java

```
public class Insert {

    public static void main(String[] args) {

    SessionFactory factory = new Configuration().configure("hibernate.cfg.xml").
                             buildSessionFactory();
      Session session = factory.openSession();
    Customer customer = new Customer();
            customer.setCustomerName("Jhonson");
            customer.setLocation("AUS");
  Transaction tx = session.beginTransaction();
            session.save(customer);
              tx.commit();
            session.close();
            factory.close();
        }
}
```

Note Before we execute the above client application, we need to create a sequence with the name new_sequence like the following,

SQL > create sequence new_sequence;

⇒ The above customId generated is a database dependent.

Q) In a POJO class of hibernate, for properties which is better to use either Primitive or Wrapper type?

A) If it is a primitive type and if we don't set value to that property then it takes its default value zero(0) will be inserted into database.

⇒ If it is wrapper type and if we don't set value to that property then it takes its default value "null" will be inserted in to database.

⇒ In a database "0" is a value and null is not a value. So a wrapper type is better than primitive type.

27/7/16

Instance states in hibernate

⇒ In hibernate, there are 3 states defined for a POJO class object in hibernate.

1) Transient state
2) Persistant state
3) detached state

⇒ Before a POJO class object enters into a session cache then the state of object is transient state.

⇒ When a POJO class object is entered into a session cache then that object is in Persistant state

⇒ When an object is removed from cache then it will be in detached state like session.close(), session.clear(), session.evict()

1) Transient state When a new object is created for a POJO class (or) when a null value is assigned to a POJO class object then that object will be in transient state.

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

* When an object is in transient state and if we do the changes to the propeties of that object then the changes are not effected on the data base.

⇒ A transient state and transient keywoid on java, both are not same.

2) Persistant state

⇒ When an object entered into cache of a session then that object will be in persistant state.

⇒ The state of an object is converted from transient to the persistant by calling the following methods.

1) save(" ");
2) persist(" ");
3) saveOrUpdate();
4) load();
5) get();

⇒ If we make any changes to an object, when it is in persistant state the changes are effected on database automatically.

3) Detached state When an object is thrown out of the session cache then the state of an object is converted from persistant to detached state.

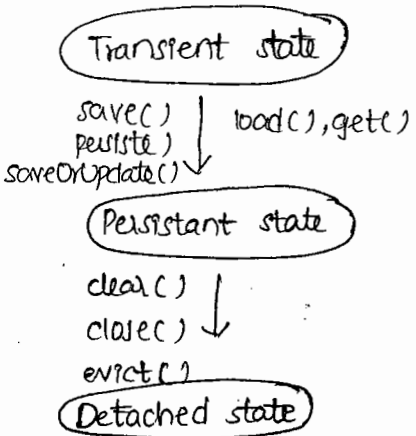
We can convert an object state from persistant to detach state by calling the following methods.

```
session.close();
session.clear();
session.evict();
```

⇒ If any changes are made on detached state object then the changes are not effected on database.

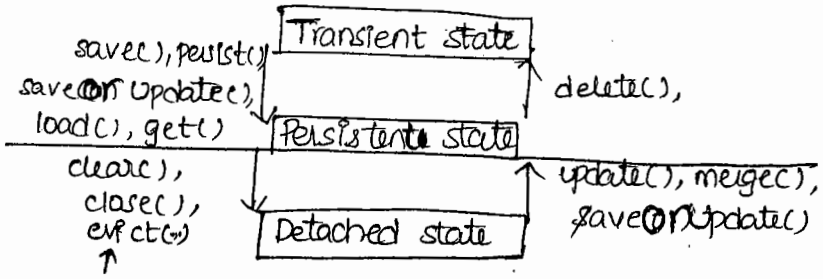
⇒ When the state of an object is transient (or) detached then the changes will not be effected on database.

⇒ But when the state is persistant the changes are effected in database.




```
Product p = new Product();
    //p → Transient state
    p.setXxx();
Transaction tx = session.beginTransaction();
        session.persist(p);
        //p → persistant state
        tx.commit();
    session.close();
    //p is in detached state.
```

28/7/15



⇒ We can convert the state of an object from detached state to persistent state, by attaching an object to the session cache.

⇒ While converting an object state from detached state to the persistent state, it is not compulsory to reattach to the same session. We can also attach to another session.

⇒ We can convert the state of an object from persistent state to transient state by permenantly deleting that object from database.

Q What is the difference between update() and merge() methods ?

A ⇒ Both methods are used for converting the state of an instance from detached state to persistant state.

⇒ When we call update() method and if already an object of same pojo class with same id exists in cache, then update() method throws "NonUnique object" Exception" because a session cache of hibernate cannot maintain two objects of same POJO class with same id.

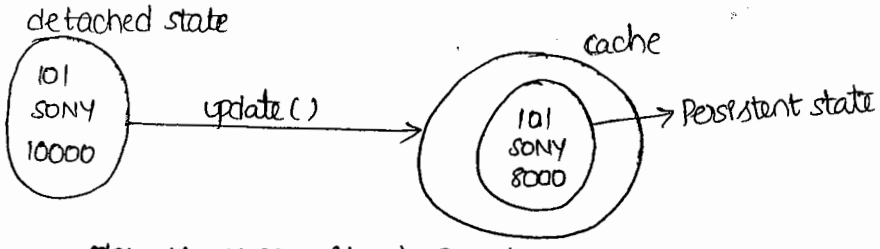
⇒ When we call merge() method and then it will first verify whether an object of same pojo class with same id exists in cache or not. If exist, it will only copy the changes from detached instance to a persistent instance in cache. If not exists, then it will directly attaches a detached instance to cache. But merge() method does not throw any exception.

Ex
```
Product P1 = (Product) session.get (Product.class, 101);
        //P1 → persistant state
session.close();
        //P1 → detached state
Session session2 = factory.openSession();
Product P2 = (Product) session2.get (Product.class, 101);
Transaction tx = session2.beginTransaction ();
```

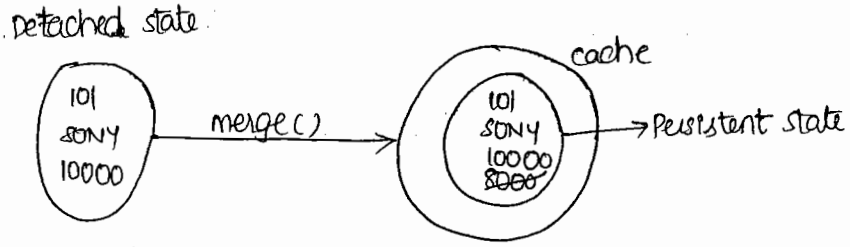
session2.update(P1); → Exception

session2.merge(P1); → works

tx.commit();



o/p: Non Unique Object Exception



o/p: No Exception (works fine)

Q Difference b/w save() & persist() methods ?

A ⇒ Persist() method will save an object in database but it does not return id of the saved object.

⇒ Save() method will save an object to database and also returns "id" of the saved object, in the form of Serializable type.

⇒ persist() method return type is void and save() method return type is serializable().

29/7/15

Q Why we need to make SessionFactory object of Hibernate as singleton?

A ⇒ In hibernate, a SessionFactory object is the only one heavy weight object. Because it stores configuration data and the mapping data of the object.project.

⇒ If our project is a desktop application then there is no need to make SessionFactory object as a singleton because, only one user can access the application at a time.

⇒ If a project is a distributed application, it means a web appln or a remoting appln then at a time multiple clients can send the request to the server appln.

⇒ If multiple SessionFactory objects are created then burden on a server increased so we need to make SessionFactory object of hibernate as singleton.

* By default, SessionFactory object of hibernate is not a singleton. It is a programmer responsibility to make it as a singleton.

⇒ The below are some scenario's, where we are going to get the problems.

Scenario(1):

⇒ We have - a servlet, it uses hibernate to go operations on database.

⇒ We have written hibernate code in service() method of servlet.

⇒ If simultaneously 40 clients have sent a request to a servlet. In this case, 40 SessionFactory objects of hibernate are created. It increases burden on a server and hence performance of an application will be decreased.

Scenario(2):

⇒ We have a project with two servlets.

⇒ In init() method of each servlet, we are creating a sessionFactory object.

⇒ Since we have two servlets, two sessionFactory objects are created.

⇒ Two SessionFactory objects increases the burden on server.

⇒ To create sessionFactory object of hibernate as a singleton, we need to create a separate utility class (helper class) and we need to define factory method in that class, to return the sessionFactory object.

⇒ All the other classes in a project will get the single SessionFactory object, by calling factory method.

Ex
```
public class HibernateUtil{
    private static SessionFactory factory;
        private HibernateUtil(){ }
Public static synchronized SessionFactory getSessionFactory(){
        If(factory==null){
    factory = new Configuration.configure("hibernate.cfg.xml").buildSessionFactory();
        }
    return factory;
    }
}
```

DAO (Data Access Object) Design Pattern

* A design pattern is not a class or interface or API or technology or a framework.

⇒ DAO is a design pattern. A design pattern is a best solution for a recursive problem.

⇒ In a realtime project, If we mix business logic (or) persistence logic then we face the following problems.

1) An application can have multiple business logics who needs database access. If persistence logic is mixed with business logic then we need to duplicate persistence logic at multiple business logic.

2) If any changes are needed to the persistence logic then we need to modify the multiple business classes.

3) We cannot test business logic separately and persistence logic separately.

⇒ As a solution for the above problems, DAO designed pattern is introduced.

⇒ A DAO design pattern says that separate the persistence logic from business logics and define persistence logic in a separate class.

This class is also called as "DAO class."

⇒ By applying DAO design pattern we will get the following benefits.

1) Multiple business logics can call a single persistence logics class. so we will get reusability.

2) We can modify persistance logic independently, by with out modifying business logics. This is called loose coupling means it separate between business logic & persistence logic

3) We can test business logic and persistance logics both independently.

31/7/15

⇒ In real-time project, a DAO design pattern is implementing by creating these files,

1) dao interface
2) dao class
3) A dao factory class.

⇒ Actual persistance logic will be defined in DAO class.

⇒ If a business class wants to call the methods of a dao class then a business class should know the dao class methods.

⇒ In real-time, we prepare a DAO interface with list of methods in dao class and that dao interface will be sent to business classes.

⇒ In order to return a dao class object to the business classes, we create a dao-factory class.

For ex:

```
//dao interface
public interface BookDAO
{
    void saveBook(Book b);
    Book readBook(int bookId);
}
```

```
//dao class
public class BookDAOImpl implements BookDAO {
    @Override
public  void saveBook(Book b)
    {
      //logic
    }
    @Override
public  Book readBook(int bookId) {
    //logic
}   }
```

```
//dao factory class
public class BDAOfactory {
    public static BookDAO getInstance()
    {
      return new BookDAOImpl();
} }
```

Creating Hibernate application of Eclipse

STEP-(1) start eclipse (D:|eclipse|eclipse.exe)

STEP-(2) In workspace lauchar, enter D:|Hibernate --> OK

STEP-(3) Click on File → New → Project → JavaProject → Next →
ProjectName : Application-1 → Finish.

STEP-(4) Right click on projectname → Build Path → Configure Build Path → Libraries →
Add External Jars button → Add all the jars required → OK.

STEP-(5) Expand Application-1 → Right click on src folder → New → Package
Name : com.sathya.hibernate.model → Finish . Similarly create 3 more

packages : com.sathya.hibernate.config
com.sathya.hibernate.dao
com.sathya.hibernate.util .

STEP-(6) Right click on model package → New → Class → Name : Book → Finish.

```
package com.sathya.hibernate.model;
Public class Book
{
  private int BookId;
  private string bookName;
  private double price;

  // setter and getter methods

}
```

STEP-(7) Right click on config package → New → File → FileName : book.hbm.xml →
Finish.

copy hibernate.cfg.xml file in to config package.

book.hbm.xml

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd ">
<hibernate-mapping>
  <class name="com.sathya.hibernate.model.Book" table="BOOK">
    <id name="bookId" column="bid">
      <generator class="increment"/>
    </id>
    <property name="bookName" column="bname" length="20" />
    <property name="price" />
  </class>
</hibernate-mapping>
```

STEP(8)

hibernate.cfg.xml

```
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
<property name="connection.username">system</property>
<property name="connection.password">tiger</property>
<property name="hibernate.dialect">org.hibernate.dialect.OracleDialect</property>
<property name="hibernate.show_sql">true</property>

<mapping resource="com/sathya/hibernate/config/book.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Note : In hibernate.cfg.xml we need to modify the mapping resource as above.

STEP(9)

```
package com.sathya.hibernate.util;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HibernateUtil{
private static SessionFactory factory;
  private HibernateUtil(){}
public static synchronized SessionFactory getSessionFactory{
    if(factory==null){
        factory=new Configuration().configure("com/sathya/hibernate/config/
                                              hibernate.cfg.xml").buildSession
                                                                 Factory();
      }
  return factory;
} //method
} //class
```

6/8/15

STEP-10) Right click on dao package → New → Interface → Name : BookDAO → Finish.

```
package com.sathya.hibernate.dao;
import com.sathya.hibernate.model.Book;
public interface BookDAO{

public void saveBook(Book b);
    Book readBook(int bookId);

}
```

STEP-11) Right click on dao package → New → class → Name : BookDAOImpl → Add → Interface Name → BookDAO → OK → Finish

```
package com.sathya.hibernate.dao;

public class BookDAOImpl implements BookDAO{

    @override
  public void saveBook(Book b){

    SessionFactory factory = HibernateUtil.getSessionFactory();

    Session session =factory.openSession();

    Transaction tx=session.beginTransaction();
          session.save(b);
          tx.commit();
          session.close();
   S.O.P("Book saved to Database");

  }
   @override

  public Book readBook(int bookId){

  SessionFactory factory = HibernateUtil.getSessionFactory();

    Session session =factory.openSource();

   Transaction tx=session.beginTransaction();

      Object o =session.get(Book.class, bookId);

      Book book = (Book)o;

        tx.commit();

      session.close();

   S.O.P("Book is read from database");
  return book;
} }//class
```

Right click on dao package → New → class → Name : BookDAOFactory → Finish.

```
package com.sathya.hibernate.dao;

public class BookDAOFactory
{
  public static BookDAO getInstance(){

     return new BookDAOImpl();
   } //method
} //class
```

STEP-12)

Right click on src folder → New → class → Name : Main → Finish

```
public class Main
{
  public static void main(String[] args)
   {
     BookDAO dao = BookDAOFactory.getInstance();
     Book b = new Book();
        b.setBookName("ThinkBook");
        b.setPrice(600);
      // call saveBook()
      dao.saveBook(b);
   S.O.P("======");
     //call readBook()
   Book book = dao.readBook(1);
    S.O.P("Book Id :" + book.getBookId());
   S.O.P("Book Name :" + book.getBookName());
   S.O.P("Price :" + book.getPrice());
  } //main
} //class
```

STEP-13) Right click on Main class → Run as → Java Application

O/P:

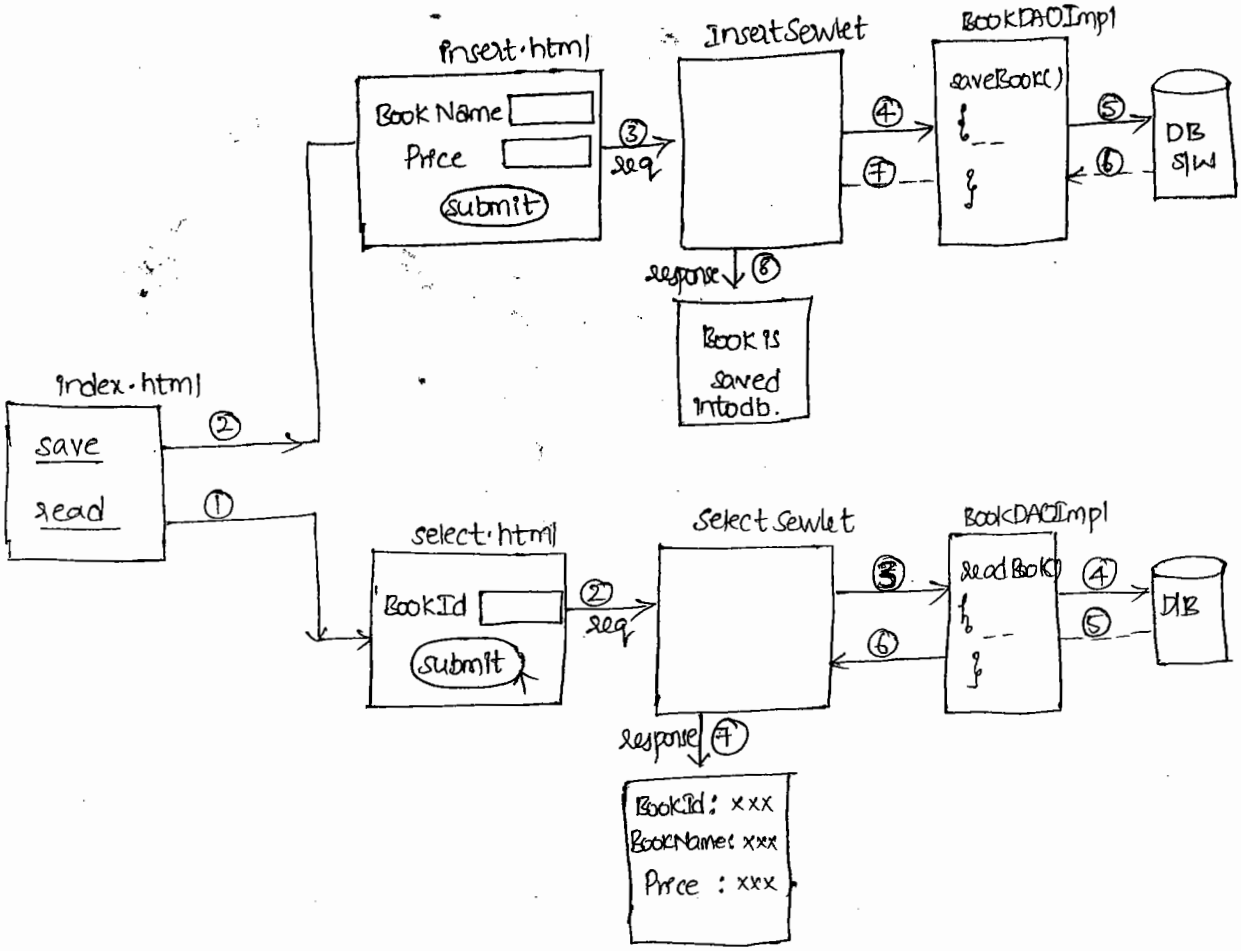
```
Book saved to Database
========
Book is read from database
Book Id : 1
BookName : ThinkBook
Price : 600.0
```

Servlet - Hibernate integration

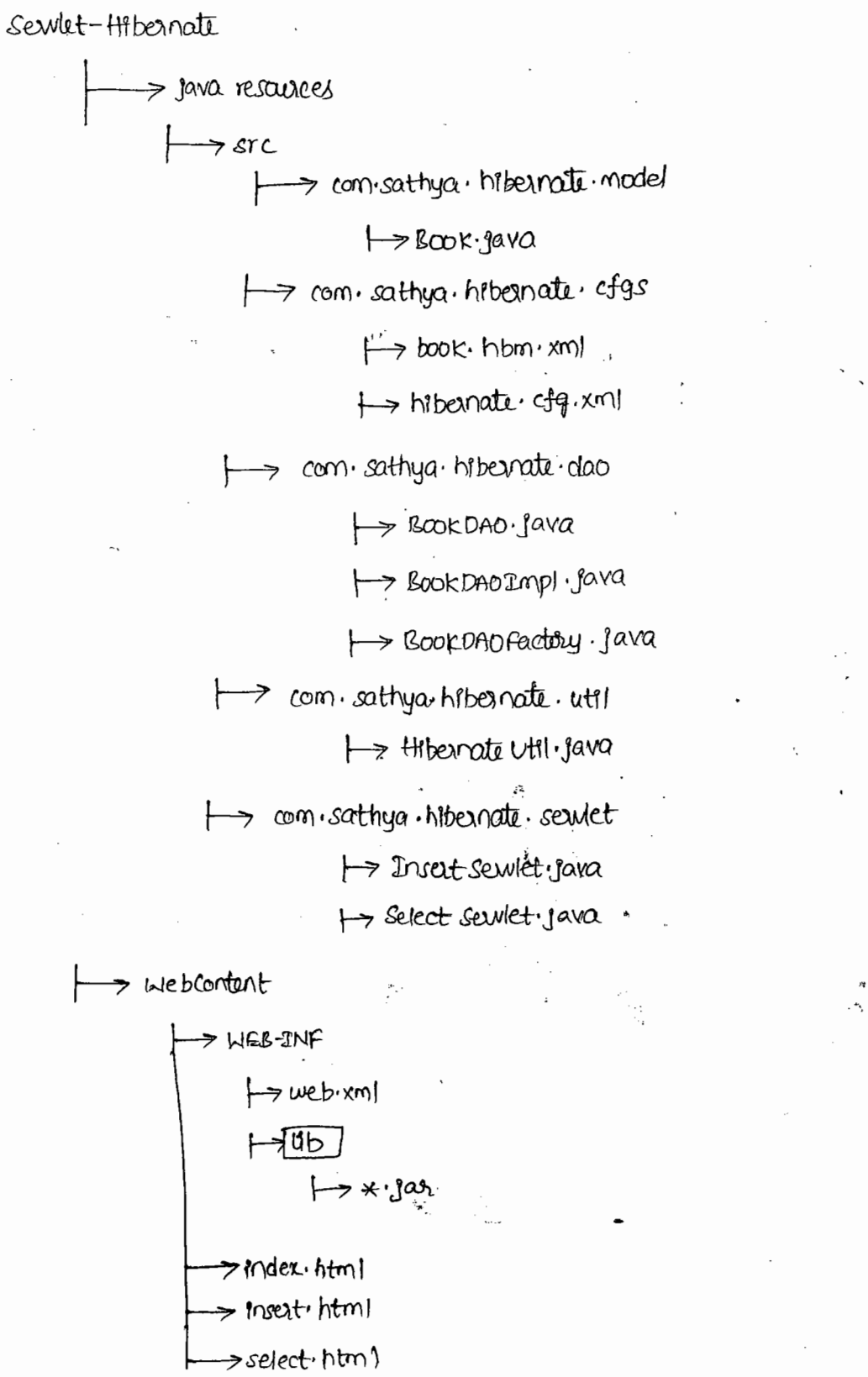
⇒ In the following example, we call methods of dao class from servlets.

⇒ In this example, we have html → servlet → dao → database communication (Hibernate).

03/08/15



Ex:


Sewlet-Hibernate
Java resources
src
com.sathya.hibernate.model
Book.java
com.sathya.hibernate.cfgs
book.hbm.xml
hibernate.cfg.xml
com.sathya.hibernate.dao
BookDAO.java
BookDAOImpl.java
BookDAOFactory.java
com.sathya.hibernate.util
HibernateUtil.java
com.sathya.hibernate.sewlet
InsertSewlet.java
SelectSewlet.java
WebContent
WEB-INF
web.xml
lib
*.jar
index.html
insert.html
select.html

STEP(1) File menu → New → Dynamic WebProject → Project Name : Sewlet-Hibernate →
select Dynamic web module version -2.5 → Finish

STEP(2) copy hibernate related jars, ojdbc6.jar & sewlet-api.jar to lib folder

STEP(3) copy model, config, dao & util packages from previous project to this current project

STEP(4) Right click on web-content folder → New → HTML File → Enter filename as index.html
→ Finish.

```
<!-- index.html -->
<html>
<h1>
<a href="insert.html"> save </a> <br>
<a href="select.html"> read </a> <br>
</h1>
</html>
```

STEP(5) Right click on web-content folder → New → HTML File → Enter filename as
insert.html → finish.

```
<!--insert.html-->
<html>
<h1> <form action="insertsrv">
Book Name : <input type=text name="bname"> <br>
Price : <input type=text name="price"> <br>
<input type=submit value="submit">
</form>
</h1>
</html>
```

STEP(6) Right click on web-content folder → New → HTML File → Enter filename as →
select.html → finish.

```
<!--select.html-->
<html>
<h1> <form action="selectsrv">
Book Id : <input type=text name="bid">
<input type=submit value="submit">
</form> </h1> </html>
```

STEP(7) Right click on src folder → New → package → Enter name → com.sathya.hibernate.
sewlet → Finish.

STEP(8) Right click on sewlet package → New → Sewlet → enter classname → InsertSewlet →
Next → Remove existing URL mapping → add buttons → enter pattern → /insertsrv →
OK → New → check doGet method → Finish.

SelectServlet.java

```
public class SelectServlet extends HttpServlet{

    Protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                    throws ServletException, IOException
    {
      //read input
    String strBookId = request.getParameter("bid");
     //wrap
      int bookId = Integer.parseInt(strBookId);
   //read dao
BookDAO dao = BookDAOfactory.getInstance();
   Book b = dao.readBook(bookId);

 PrintWriter out = response.getWriter();
     if(b==null)
     {
      out.println("Sorry the given bookid doesn't exist in Database In");
     }
    else
     {
      out.println("<h1>");
                ("Book Id : "+b.getBookId());
                ("<br>");
                (" Book Name : "+ b.getBookName());
                 ("<br>");
                ("Price : "+b.getPrice());
                ("</h1>");
        }
     out.close();
  }//end of doGet();

}//end of class
```

STEP click on window → show view → Servers → Right click in server view → New → server → expand Apache → Tomcat v7.0 Server → Next → browse → select Tomcat server folder → Finish.

STEP Right click on project name (Servlet-hibernate) → Run as → Run on server → Next → Finish.

InsertSewlet.java

```
public class InsertSewlet extends HTTPSewlet
{
  protected void doGet( HttpSewletRequest request, HttpSewletResponse response )
                        throws SewletException, IOException
  {
    // read input
  String bookName = request.getParameter("bname");
  String strPrice = request.getParameter("price");
  double price = Double.parseDouble(strprice);
        // create POJO object
     Book book = new Book();
        book.setBookName(bookName);
        book.setPrice(price);
       // read dao object
  BookDAO dao = BookDAOFactory.getInstance();
  PrintWriter out = response.getWriter();
     try {
          dao.saveBook(book);
          out.println("<h1> Books saved into DB </h1>");
        }
     catch (Exception e){
       out.println("<h1> Sorry, Problem in saving </h1>");
      }
     out.close();
    } // end of doGet()
  } // end of class
```

STEP (7) Right click on sewlet package → New → Sewlet → enter class name → SelectSewlet → Next → Remove url mapping → [add] → enter pattern → [/select-srv] → [OK] → [Next] → check [doGet] method → Finish.

04/08/15

Inheritance Mapping | Inheritance strategies

⇒ In a hibernate application, if there are multiple pojo classes and if they have common properties, then to get reusability we apply inheritance.

⇒ Common properties, we separate and we create in super class and we extend that super class to multiple subclasses.

Ex: We have two pojo classes in a project called "CreditCard" and "Cheque" like the following.

```
public class CreditCard {
private int paymentId;
private double paidAmount;
private Date paymentDate;
private int cardNumber;
private String cardType;

//setters and getters methods

}
```

```
public class Cheque {
private int paymentId;
private double paidAmount;
private Date paymentDate;
private int chequeNumber;
private String chequeType;

//setters and getters methods

}
```

⇒ In the above two pojo classes, there are 3 common properties. So in order to get reusability, we apply inheritance by separating common properties to a super class like the following,

```
public class Payment {
private int paymentId;
Private double paidAmount;
Private Date paymentDate;
// setters and getters methods
}
```

```
public class CreditCard extends Payment{
    private int cardNumber;
    private String cardType;
    //setters and getters Methods
}
```

```
public class cheque extends Payment{
    private int chequeNumber;
    private String chequeType;
    //setters and getters Methods
}
```

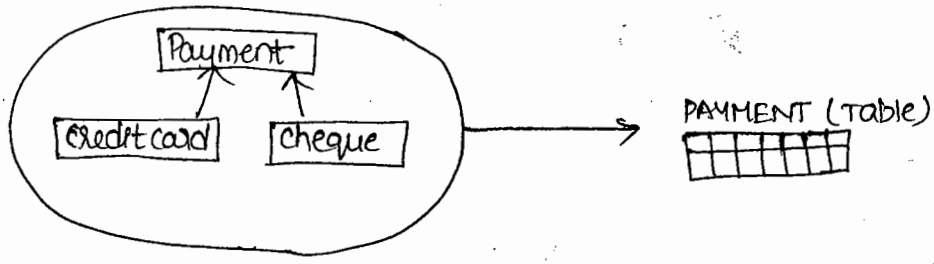
⇒ Here "Payment" class is abstract and "CreditCard, Cheque" are concrete classes.

05/08/15

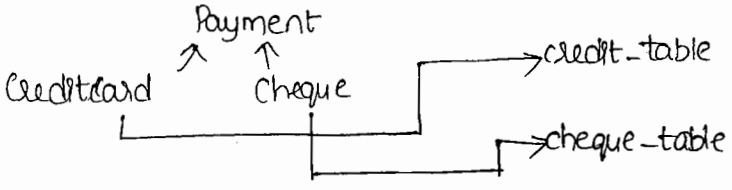
⇒ Hibernate has provided 3 inheritance strategies, to map the classes of hierarchy to database tables.

1) Table per class
2) Table per concrete class
3) Table per subclass

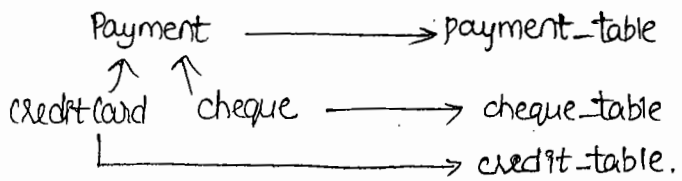
⇒ We need to choose table per class strategy, if we want to map all the classes of hierarchy to a single table of database.



⇒ If we want to map each concrete class of hierarchy to a separate tables of database then we need to select 'table per concrete class' strategy



⇒ We we want to map each class of hierarchy to a seperate table of database then we need to select table per subclass strategy.



1) Table per class strategy [mapping file]

⇒ In this inheritance strategy, all concrete classes objects of hierarchy will be stored in a single database table.

⇒ In table, one additional column is also required for storing discriminator value of a concrete class. that additional column is called a 'discriminator column'.

⇒ A discriminator value is to identify a row in that table belongs to which concrete class object.

⇒ In a hbm-file, to tell the hibernate that table per class is the inheritance strategy selected, we need to configure <subclass> tag under <class> tag.

⇒ To tell hibernate about name of the discriminator column, we configure <discriminator> tag immediately after <id> tag.

⇒ For example, if we select table per class strategy for the payment example then in database we need a payment table like the following,

PAYMENT (TABLE)

payid ↑ primary key	amount	pdate	ccno	cctype	chno	chtype	pmode ↑ discriminator column

⇒ We need to construct the hbm file like the following,

payment.hbm.xml

```
<hibernate-mapping>
 <class name="payment" table="payment">
  <id name="paymentId" column="payid" />
   <discriminator column="pmode" type="string" length="6" />
   <property name="amount" />
   <property name="paymentDate" column="pdate" type="date" />
   <subclass name="CreditCard" discriminator-value="cc">
     <property name="cardNumber" column="ccno" />
     <property name="cardType" column="cctype" />
   </subclass>
  <subclass name="Cheque" discriminator-value="CH">
    <property name="chequeNumber" column="chno" />
    <property name="chequeType" column="chtype" />
  </subclass>
 </class>
</hibernate-mapping>
```

Project Name : TablePerClass

- src
  - com.sathya.hibernate.model
    - Payment.java
    - CreditCard.java
    - cheque.java
  - com.sathya.hibernate.cfgs
    - hibernate.cfg.xml
    - payment.hbm.xml
  - com.sathya.hibernate.dao
    - PaymentDAO.java, PaymentDAOImpl.java
    - PaymentDAOFactory.java
  - com.sathya.hibernate.util
    - HibernateUtil.java
  - default
    - Main.java

06/08/15

// Payment.java

```
public abstract class Payment{
    private int paymentId;
    private double amount;
    private Date paymentDate;

    public int getPaymentId(){
        return paymentId;
    }
    public void setPaymentId(int paymentId){
        this.paymentId = paymentId;
    }
    public double getAmount(){
        return amount;
    }
    public void setAmount(double amount){
        this.amount = amount;
    }
    public Date getPaymentDate(){
        return paymentDate;}
    public void setPaymentDate(Date paymentDate){
        this.paymentDate = paymentDate;
    }
}
```

CreditCard.java

```
public class CreditCard extends Payment{
    private int cardNumber;
    private String cardType;
    public int getCardNumber(){
        return cardNumber;
    }
    public void setCardNumber(int cardNumber){
        this.cardNumber = cardNumber;}
    public String getCardType(){
        return cardType;}
    public void setCardType(String cardType){
        this.cardType = cardType;}
```

```
// Cheque.java
public class Cheque extends Payment {

    private int chequeNumber;

    private string chequeType;

        public int getChequeNumber() {
                return chequeNumber; }
        public void setChequeNumber(int chequeNumber) {
                this.chequeNumber = chequeNumber; }
        public string getChequeType() {
                return chequeType; }
        public void setChequeType(string chequeType) {
                this.chequeType = chequeType; }
}
```

hibernate.cfg.xml

```
<mapping resource = "com/sathya/hibernate/config/payment.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

payment.hbm.xml

```
<hibernate-mapping>
<class name = "com.sathya.hibernate.model.Payment" table = "payment" >
  <id name = "paymentId" column = "payid" type = "int" />
  <discriminator column = "pmode" type = "string" length = "6" />
   <property name = "amount" type = "double" />
   <property name = "paymentDate" column = "pdate" type = "date" />
  <subclass name = "com.sathya.hibernate.model.CreditCard" discriminator-value = "cc" >
      <property name = "cardNumber" column = "ccno" type = "int" />
      <property name = "cardType" column = "cctype" type = "string" length = "9" />
  </subclass>
  <subclass name = "com.sathya.hibernate.model.Cheque" discriminator-value = "CH" >
    <property name = "chequeNumber" column = "chno" type = "int" />
    <property name = "chequeType" column = "chtype" type = "string" length = "9" />
  </subclass>
</class>
</hibernate-mapping>
```

PaymentDAO.java

```
package com.kpbaya.hibernate.doo;
public interface PaymentDAO{

    void saveCard(CreditCard card);
    void saveCheque(Cheque cheque);

}
```

PaymentDAOImpl.java.

```
public class PaymentDAOImpl implements PaymentDAO{

    @Override
  public void saveCard(CreditCard card)
  {
    SessionFactory factory = HibernateUtil.getSessionFactory();
       Session session = factory.openSession();
    Transaction tx = session.beginTransaction();
     session.save(card);
      tx.commit();
     session.close();
   S.o.p("card details saved");
  }
}

   @Override
   public void saveCheque(Cheque cheque){
      SessionFactory factory = HibernateUtil.getSessionFactory();
           Session session = factory.openSession();
       Transaction tx= session.beginTransaction();
              session.save(cheque);
              tx.commit();
              session.close();
            S.o.p("cheque details saved");
  }
}
```

```
Package com.sathya.hibernate.dao;

public class PaymentDAOFactory {

    public static PaymentDAO getInstance()
    {
        return new PaymentDAOImpl();
    }

}

public class Main {

    public static void main(String[] args) {

        PaymentDAO dao = PaymentDAOFactory.getInstance();
            //call saveCard()
        CreditCard card = new CreditCard();
            card.setPaymentId(11011);
            card.setAmount(8000);
            card.setPaymentDate(new java.util.Date());
            card.setCardNumber(321567);
            card.setCardType("MASTER");
          dao.saveCard(card);
      S.O.P("======");
        //call saveCheque()
      Cheque cheque = new Cheque();
          cheque.setPaymentId(22022);
          cheque.setAmount(5000);
          cheque.setPaymentDate(new java.util.Date());
          cheque.setChequeNumber(56022);
          cheque.setChequeType("older");
           dao.saveCheque(cheque);
       S.O.P("======");
    }
}
```

op: `select * from payment;`

payId	pmode	amount	pdate	ccNO	cctype	chno	chtype
11011	CC	8000	06-AUG-15	321567	MASTER	null	null
22022	CH	5000	06-AUG-15	null	null	56022	older

2) Table per Concrete class strategy

⇒ We select this inheritance strategy, when we want to map each concrete class of hierarchy to separate tables in the database.

⇒ In table per concrete class strategy, discriminator-column is optional.

⇒ In hbm file, we need to configure <union-subclass> tag under <class> tag, to tell hibernate that table per concrete class strategy is selected.

```
<!--payment.hbm.xml-->
<hibernate-mapping>
 <class name="com.sathya.hibernate.model.Payment">
   <id name="paymentId" column="payId" type="int"/>
   <property name="amount" type="double"/>
   <property name="paymentDate" type="date" column="pdate"/>

 <union-subclass name="com.sathya.hibernate.model.CreditCard" table="credit_table">
   <property name="cardNumber" column="ccno" type="int"/>
   <property name="cardType" column="cctype" type="string" length="9"/>
 </union-subclass>
 <union-subclass name="com.sathya.hibernate.model.Cheque" table="cheque_table">
   <property name="chequeNumber" column="chno" type="int"/>
   <property name="chequeType" column="chtype" type="string" length="9"/>
 </union-subclass>

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

o/p:- select * from credit_table;

PAYID	AMOUNT	PDATE	CCNO	CCTYPE
11011	8000	06-AUG-15	321567	MASTER

select * from cheque_table;

PAYID	AMOUNT	PDATE	CHNO	CHTYPE
22022	5000.	06-AUG-15	51098	order

3) Table per subclass strategy

⇒ In this inheritance strategy, we map each class of hierarchy to a seperate table in the database.

⇒ In database, we need parent table and child tables. To get relation between the data of two tables, we need a foreign key column ~~of~~ also in the child table.

⇒ In mapping file, we need to add <joined-subclass> tag under <class> tag, to tell the hibernate that table per subclass strategy is selected.

⇒ To tell foreign key column name to hibernate, we need to configure <key> tag under <joined-subclass> tag.

⇒ In table per subclass strategy also, discriminator column is optional.

```
<!--payment-hbm.xml -->
<hibernate-mapping>
 <class name="com.sathya.hibernate.model.Payment" table="payment">
     <id name="paymentId" column="payid" type="int" />
     <property name="amount" column="double" />
     <property name="paymentDate" column="pdate" type="date" />
  <joined-subclass name="com.sathya.hibernate.model.creditCard" table="credit-tab">
    <key column="pid" type="int" />
  <property name="cardNumber" column="ccno" type="int" />
  <property name="cardType" column="cctype" type="string" length="10" />
 </joined-subclass>
 <joined-subclass name="com.sathya.hibernate.model.cheque" table="cheque_table">
    <key column="pid" type="int" />
  <property name="chequeNumber" column="chno" type="int" />
  <property name="chequeType" column="chtype" type="string" length="9" />
  </joined-subclass>
 </class>
</hibernate-mapping>
```

dp

select * from payment;

PAYID	AMOUNT	PDATE
11011	8000	06-AUG-15
22022	5000	06-AUG-15

select * from credit-table

PID	CCNO	CCTYPE
11011	821567	MASTER

select * from cheque_table;

PID	CHNO	CHTYPE
22022	53098	order

Note

1) Primary key column & foreign key column name may be same or may not be same.
2) The datatypes of primary key and foreign key column must be same.
3) Foreign key column size must be greater than or equal to primary key column size.

Component Mapping

⇒ Component Mapping is used, when there is a "has-a" relationship between the POJO classes.

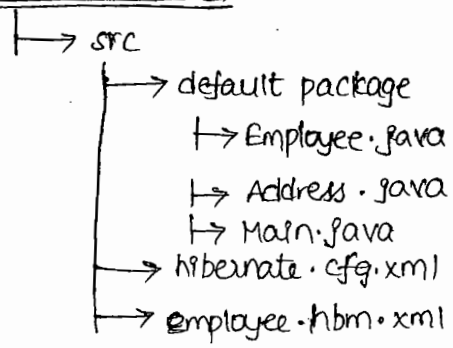
⇒ In component Mapping, object of a class will be stored as a value of another class object.

⇒ In "has-a" relationship, we call the first class as a dependent and second class as a dependency.

⇒ If we want to save a dependent object along with its dependency object in a single table of database then we need to use Component Mapping of hibernate.

⇒ In hbm file, we write <component> tag under <class> tag, to tell the hibernate that we are using component mapping.

Component-Mapping



Employee

```
public class Employee {
    private int employeeId;
    private string employeeName;
    private Address addr;

public int getEmployeeId() {
        return employeeId; }

public void setEmployeeId() {
        this.employeeId = employeeId;
            }
```

```
    public String getEmployeeName(){
            return employeeName;}
    public void setEmployeeName(String employeeName){
                this.employeeName = employeeName;}
    public Address getAddr(){
            return addr;}
    public void setAddr(Address addr){
                this.addr = addr;}
}
```

Address.java

```
public class Address {
    private String hno, street, city;
    public String getHno(){
            return hno;}
    public void setHno(String hno){
            this.hno = hno;}
    public String getStreet(){
            return street;}
    public void setStreet(String street){
            this.street = street;}
    public String getCity(){
            return city;}
    public void setCity(String city){
            this.city = city;
            }
}
```

employee.hbm.xml

```
<hibernate-mapping>
    <class name="Employee" table="employee">
     <id name="employeeId" column="empid" />
     <property name="employeeName" column="empname" length="10" />
    <component name="addr" class="Address">
     <property name="hno" length="10" />
     <property name="street" length="12" />
     <property name="city" length="12" />
    </component>
</class>
</hibernate-mapping>
```

Main.java

```
public class Main {
    public class void main(String[] args) {
        //Get Session object
        Session ses = HibernateUtil.getSession();
        // Save objects
        Transaction tx=null;
        try{
            tx = ses.beginTransaction();
Address addr1 = new Address("10/B", "Rama Street", "Hyderabad");
Employee emp1=new Employee();
    emp1.setEmployeeId(1003);
    emp1.setEmployeeName("Rafa");
    emp1.setAddress(addr1);
Address addr2 = new Address("1/A", "school street", "Vijayawada");
Employee emp2 = new Employee();
        emp2.setEmployeeId(1004);
        emp2.setEmployeeName("Ravi");
        emp2.setAddress(addr2);
    // save objects
    ses.save(emp1);
    ses.save(emp2);
    tx.commit();
}//try
catch(Exception e){
        tx.rollback();
        e.printStackTrack();
    }
//close session
ses.close();
}
}
```

08/08/15

How to configure the composite id in Hibernate?

⇒ In a database, we use a primary key, to identify one row uniquely among multiple rows stored in a table.

⇒ In most of the cases, a single column of table is suitable to use it as a primary key. But in some cases, we need combination of 2 or more columns is needed to uniquely identify a row. It is called composite primary key.

⇒ In database, a table can have maximum a single primary key.

⇒ In Hibernate, to map the properties of POJO class to a composite primary key, we use <composite-id> tag.

Example

→ Let us say, we have a table Appolo, which stores total sale of a medicine from each outlet in the central table of a particular date. A sample data is like the following.

Appolo

outid	location	medicine	pdate	price
1001	A.pet	Dolo650	8-AUG-15	30
1005	P. Gutta	Corex	8-AUG-15	90
1002	Sr Nagar	Dolo650	8-AUG-15	50
1001	A.pet	Dart	8-AUG-15	90
1002	Sr Nagar	Nimul	9-AUG-15	170

→ In the above table, each individual column have duplicate value. so a single column cannot act as a primary key.

→ We can use outid + medicine + pdate columns combination as a primary key. Because we cannot find a duplicate row with these columns combination. so it is a composite-primary key)

→ A POJO class and a mapping file, to map in appolo table will be like the follow

```
public class Appolo {
    private int outletId;
    private String location;
    private String medicine;
    private Date purchaseDate;
    private double price;
} ⇒ setters & getters
```

Appolo.hbm.file

```
<hibernate-mapping>
 <class  name="Appolo"  table="Appolo">
    <composite-id>
      <key - property  name="outletId"  column="outid" />
      <key- property   name="medicine" />
      <key- Property   name="purchaseDate"  column="pdate" />
    </composite-id>
   <property   name="location" />
   <property   name="price" />
 </class>
</hibernate-mapping>
```

Note

⇒ In every hbm file, there must be <id> tag or <composite-id> tag.

⇒ We cannot apply hibernate generators to <composite-id>. Bydefault assigned generator will be applied on <composite-id>.

Bulk Operations

⇒ By calling save() method, update(), delete(), load(), get() and etc.., we can perform CURD operations on a single object at a time.

⇒ If we want to perform CURD operations on multiple objects at a time, we use bulk operation techniques of hibernate.

⇒ The 3 bulk operation techniques are,

(i) HQL (hibernate query language)
(ii) Criteria
(iii) Native SQL

HQL (Hibernate Query Language)

⇒ It is own Query language of Hibernate it means we no need of changing a query, while connecting with another database.

⇒ HQL looks like SQL only so it is easy to learn.

⇒ To construct HQL Queries, we use variable names in place of column names and classname in place of table name. Call HQL as object oriented form SQL using HQL, we can perform both select and non-select operations on a database.

HQL for select:

Ex(1): SQL : select * from emp;
HQL : from Employee e;

⇒ In hibernate, reading a complete row is called reading a complete entity.

⇒ To read a complete entity (or entities, we need to begin HQL command with from keyword.

⇒ Ex(2)

SQL : select empno, sal from emp;
HQL : select e.employeeName, e.employeeSalary from Employee e;

⇒ reading the values of a specific columns is called reading Partial Entity.

⇒ In hibernate, to read a partial entity (or) a entities, a query begins with select keyword.

Ex3

SQL : select * from emp where deptno = ?
HQL : from Employee e where e.deptNumber = ?
(or)
from Employee e where e.deptNumber : P1

⇒ In order to set runtime values into HQL command, we can use either index parameter(?) (or) named Parameter (:P1)

Note We can use all clauses like between, having, group by, order by, etc... In HQL commands.

How to run HQL select operation

⇒ If we want to execute HQL command, first we need a Query object.

⇒ on Query object, we call list() method, to run the select command of HQL. In hibernate, Query is a predefined interface and we can get its implementation class object, by calling createQuery() method of Session Interface.

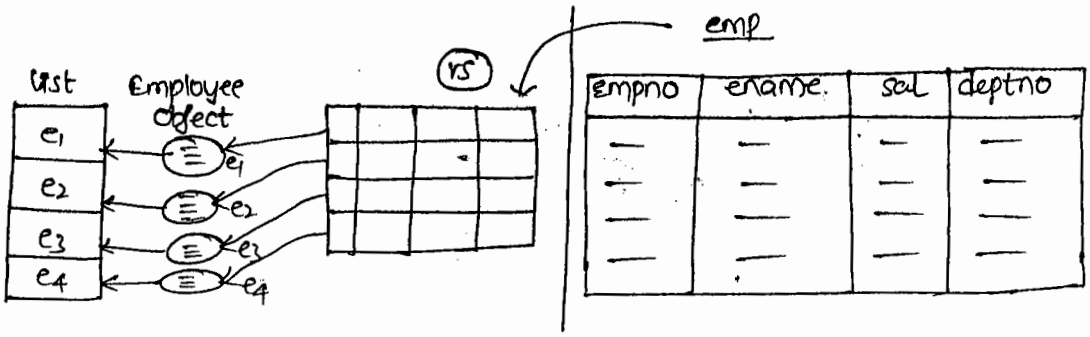
Ex: Query qry = session.createQuery("from Employee e");
List list = qry.list();

⇒ When we execute select operation of HQL then hibernate stores one of the following three in a list object.

1) If the HQL command is to read complete entities then hibernate stores objects of POJO class in List.
2) If HQL command is to read partial entities then hibernate stores Object[]'s in list.
3) If the HQL command is to read partial entities with a single column then hibernate stores objects of that property type in List.

Ex(1)

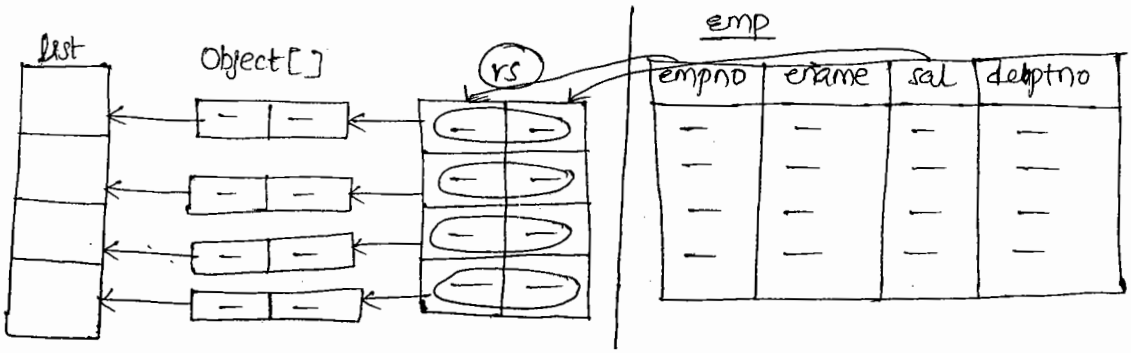
```
Query query = session.createQuery("from Employee e");
List list = query.list();
```




```
Iterator itr = list.iterator();
while(itr.hasNext()){
Employee e = (Employee) itr.next();
}
```

Ex(2)

```
Query query = session.createQuery("select e.employeeNumber, e.employeeSalary
                                   from Employee e");
List list = query.list();
```

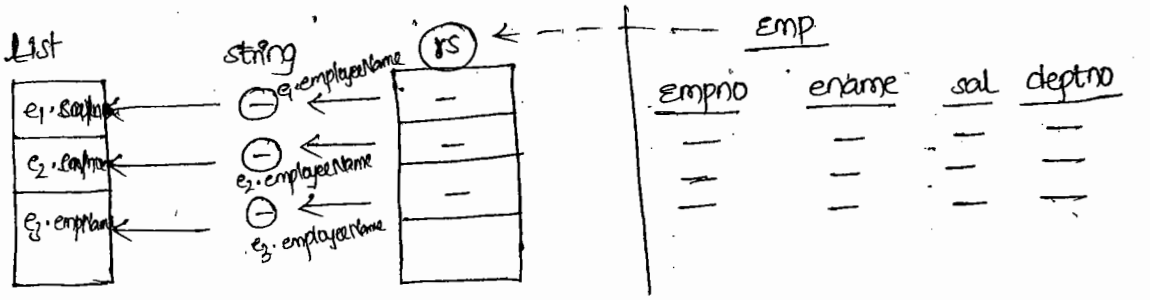



```
Iterator itr = list.iterator();
while(itr.hasNext())
{
Object[] obj = (Object[]) it.next();
S.o.p("obj[0]+" "+obj[1]+" "+obj[2]+" ");
}
```

11/08/15

example3.

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

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




```
Iterator it = list.iterator();
    while(it.hasNext())
    {
      String str = (String) it.next();
       S.O.P(str);
    }
```

HQL for non-select operations

⇒ update and delete operations of HQL are similar to SQL. But insert operation is different.

⇒ In HQL, insert operation is used to copy the reads from one table to another table

⇒ Eg1 :

sql → update emp set sal=9000 where deptno=10

HQL → update Employee e set e.employeeSalary = 9000 where e.deptNumber=10

Eg2:

SQL → delete from emp where sal>10000

HQL → delete from Employee e where e.employeeSalary >10000

Eg3:

properties of Test class

HQL → insert into Test t (empid, empname, empsal, deptno) select e.employeeNumber, e.employeeName, e.employeeSalary, e.deptNumber from Employee e

⇒ The above insert command of HQL, copies records from emp table to test table in database.

⇒ In the above insert command, Test and Employee are two POJO classes.

⇒ When writting insert command of HQL, the proper no. of properties of source class and no. of properties of destination class must be same.

How to run non-select of HQL

⇒ While executing non-select operations, a transaction is mandatory.

⇒ To run non-select operation of HQL, follow the below steps.

a) create a Query object

b) begin a Transaction

c) call executeUpdate()

d) commit transaction.

Eg:

```
① Query query = session.createQuery("update Employee e set e.employeeSalary
                                      = 9000 where e.deptNumber=10");
② Transaction tx = session.beginTransaction();
③     int i = query.executeUpdate();
④         tx.commit();
```

HQL Example

```
|-> src
    |-> com.sathya.hibernate.model;
        |-> Employee.java

    |-> com.sathya.hibernate.config
        |-> hibernate.cfg.xml
        |-> employee.hbm.xml
    |-> com.sathya.hibernate.dao
        |-> EmployeeDAO.java
            EmployeeDAOImpl.java
            EmployeeDAOFactory.java
    |-> com.sathya.hibernate.util
        |-> HibernateUtil.java
|-> default
    |-> Main.java
```

Model

```
public class Employee{

    private int employeeNumber;
    private string employeeName;
    private int employeeSalary;
    private int deptNumber;
  // setters and getters

    public int getEmployeeNumber(){
            return employeeNumber; }

    public void setEmployeeNumber( int employeeNumber){
            this.employeeNumber = employeeNumber; }

    public string getEmployeeName(){
            return employeeName; }

    public void setEmployeeName( string employeeName){
            this.employeeName = employeeName; }

    public int getEmployeeSalary(){
            return employeeSalary; }

    public void setEmployeeSalary ( int employeeSalary){
            this.employeeSalary = employeeSalary; }

    public int getDeptNumber(){
            return deptNumber;
            }

    public void setDeptNumber(int deptNumber){
        this.deptNumber = deptNumber;
            }

}
```

dao

```
public interface EmployeeDAO {

    void findEmployeesByDeptno (int deptno);
    void updateEmployeesByDeptno(int deptno);

}

public class EmployeeDAOImpl implements EmployeeDAO {

  @override
 public void findEmployeesByDeptNO ( int deptno ) {

   SessionFactory factory = HibernateUtil.getSession();
     Session session = factory.openSession();
    Query query = session.createQuery("from Employee e where e.deptNumber = ? ");
       query.setParameter(0, deptno);
     List list = query.list();
    Iterator it = list.iterator();
    while(it.hasNext())
     {
        Employee e = (Employee) it.next();
        s.o.p(e.getEmployeeNumber() + "  " + e.getEmployeeName() + "  " +
     }            e.getEmployeeSalary() + "  " + e.getDeptNumber());

    session.close();

  }

  @override
public void updateEmployeesByDeptno(int deptno) {

     SessionFactory factory = HibernateUtil.getSession();
      Session session = factory.openSession();
    Query query = session.createQuery("update Employee e set e.employeeSalary=8709
                                       where e.deptNumber = ? ");
       query.getParameter(0, deptno);
    Transaction tx = session.beginTransaction();
       int i = query.executeUpdate();
        tx.commit();
  s.o.p(i + " rows updated "); session.close(); } }
```

EmployeeDAOfactory.java

```
public class EmployeeDAOfactory {

    public static EmployeeDAO getInstance()
    {
        return new EmployeeDAOImpl();
    }

}
```

Main.java

```
public class Main{

    public static void main(String[] args){

        EmployeeDAO dao = EmployeeDAOfactory.getInstance();

            dao.findEmployeesByDeptno(20);

        S.O.P("====");

            dao.updateEmployeesByDeptno(10);

    }
```

12/08/15

2) Criteria API (for select operation)

⇒ Criteria is another technique to perform bulk operations.

⇒ With criteria, we can only perform select operation.

⇒ Even with HQL, by creating select commands we can read the data from database.

⇒ If we directly write HQL select queries then we also need to tune the query, for better performance.

⇒ We can read same data from database by constructing a query in multiple ways. But performance is important.

⇒ To read data with in a less amount of time, we need to apply query tuning.

⇒ As a java developers we are not having much knowledge on query tunning. so hibernate has provided criteria API, which prepares a tuned query on behalf of a programmer and then executes that query on the database.

⇒ Query tuning is only required for reading the data from the database but not required for non-select operations. So criteria api can be used only for select operations.

⇒ To read either complete entities or partial entities, first we need a Criteria object.

⇒ We can get the Criteria object by calling createCriteria() of Session Interface.

⇒ While creating a criteria object, we need to pass Class object of POJO class as a parameter.

Reading complete entities/entity

```
Criteria criteria = session.createCriteria(Employee.class);
List list = criteria.list();
```
↓
POJO objects.

Reading complete entity/entities with condition

⇒ To add condition to Criteria, we need a Criterion object.

⇒ Criterion is an interface and we can get a Criterion object by calling static methods of Restrictions class.

⇒ Restrictions is not an implementation class of Criterion interface.

⇒ Criterion interface and Restrictions class both are under same package
org.hibernate.criterion;

⇒ The following criteria reads employees working in deptNumber '20'.

```
Criteria  criteria = session.createCriteria(Employee.class);
Criterion  c1 = Restrictions.eq("deptNumber", 20);
        criteria.add(c1);
List  list = criteria.list();
        ↓
    contains pojo object whose deptno = 20;
```

⇒ The following criteria reads employees working in deptNumber = '20' and whose salary > 1000;

```
Criteria  criteria = session.createCriteria(Employee.class);
Criterion  c1 = Restrictions.eq("deptNumber", 20);
Criterion  c2 = Restrictions.eq("employeeSalary", 10000);
Criterion  c3 = Restrictions.and(c1, c2);
        criteria.add(c3);
List  list = criteria.list();
        ↓
    contains pojo object whose deptno = 20 & salary > 1000;
```

Reading partial entity/entities with condition

⇒ To read partial entity/entities, we need a Projection object for each property.
⇒ We can get a Projection object by calling static method of Projections class.
⇒ Projections is not an implementation class of Projection Interface.
⇒ Add Projection objects to a ProjectionList and then set ProjectionList object to criteria.

The following criteria reads employees with only no & name having sal > 1000 from DB.

```
Criteria  criteria = session.createCriteria(Employee.class);
Projection  p1 = Projections.property("employeeNumber");
Projection  p2 = Projections.property("employeeName");
ProjectionList  plist = Projections.projectionList();
```

```
plist.add(P1);
plist.add(P2);
criteria.setProjection(plist);
Criterion c1 = Restrictions.eq("deptNumber", 20);   } condition.
criteria.add(c1);
List list = criteria.list();
```
↑ contains pojo class object array

13/08/15

⇒ Suppose if we want to read a single column then we no need of ProjectionList obj. We can directly set a Projection object to criteria.

```
Criteria criteria = session.createCriteria(Employee.class);
Projection p1 = Projections.property("employeeName");
criteria.setProjection(p1);
List list = criteria.list();
```
↓ It contains string objects.

Reading partial entity/ entities with condition

⇒ To read partial entities, we need to set a Projection(or) ProjectionList object to criteria and to apply the condition, we need to set a Criterion object to a criteria.

```
Criteria criteria = session.createCriteria(Employee.class);
Projection p1 = Projections.property("employeeNumber");
Projection p2 = Projections.property("employeeName");
ProjectionList plist = Projections.projectionList();
plist.add(p1);
plist.add(p2);
criteria.setProjection(plist);
Criterion c1 = Restrictions.ge("employeeSalary", 5000);
criteria.add(c1);
List list = criteria.list();
```
↓ contains object array.

Reading aggregate values

⇒ To read aggregate values like count of rows (or) maximum value of a column (or) minimum value of a column (or) average of a column values etc.. then we need to use projection concept of hibernate.

⇒ Projection can be used in two cases

1) To read partial entities
2) To read aggregate values

for example

```
Criteria criteria = session.createCriteria(Employee.class);
Projection p1 = Projections.rowCount();
Projection p2 = Projections.sum("employeeSalary");
Projection p3 = Projections.max("employee_Salary");
ProjectionList plist = Projections.projectionList();
        plist.add(p1);
        plist.add(p2);
        plist.add(p3);
    criteria.setProjection(plist);
List list = criteria.list();
```

↓

single object array.

Object[] | rowcount | sum | max |

```
for(i=0; i<list.size(); i++){
  Object[] object = (Object[])list.get(i);
  }  s.o.p(object[0]);
     s.o.p(object[1]);
     s.o.p(object[2]);
```

Reading entities in Soting order

⇒ To read entities in a soting order, then we need to add an object of Order class to criteria object.

⇒ We can get an object of Order class by calling either asc() (or) desc() [static methods] of Order class.

For example

⇒ If we want to read employees in ascending order of the salaries, then we need to use the following code.

```
Criteria criteria = session.createCriteria(Employee.class);
    Order order = Order.asc("employeeSalary");
        criteria.addOrder(order); || criteria.addOrder(order);
    List list = criteria.list();
```

Criterion (I)
Restriction (C)
Projection (I)
Projections (C)
ProjectionList (I)
Order (C)

} org.hibernate.criterion

Pagination Methods

⇒ These methods are used to read only a specific records from the database

1) setFirstResult()
2) setMaxResults()

⇒ setFirstResult() is to set the starting row index.

⇒ setMaxResults() is to set no.of rows.

For example

```
Criteria criteria = session.createCriteria(Employee.class);
        criteria.setFirstResult(2); || from third row (index starts from 0)
        criteria.setMaxResults(5); || 5 rows (3,4,5,6,7 rows).
    List list = criteria.list();
```

Example program on criteria api for Bulk Operation (select operations)

employee.hbm.xml

```
<hibernate-mapping>
 <class name="com.s.h.m.EmpDetails" table="Employee">
 <id name="no" column="EID" />
 <property name="fname" column="FIRSTNAME" length="20" />
 <property name="lname" column="LASTNAME" length="20" />
 <property name="mail" column="EMAIL" length="20" unique="true" not-null="true" />
 </class>
</hibernate-mapping>
```

EmpDetails.java

```
public class EmpDetails {
    private int no;
    private String fname, lname, mail;

    public EmpDetails(){
        s.o.p("EmpDetails :0-param constructor");
    }

    public int getNo(){
        return no;
    }

    public void setNo(int no){
        this.no = no;
    }

    public String getFname(){
        return fname;
    }

    public void setFname(String fname){
        this.fname = fname;
    }

    public String getLname(){
        return lname;
    }

    public void setLname(String lname){
        this.lname = lname;
    }

    public String getMail(){
        return mail;
    }

    public String setMail(String mail){
        this.mail = mail;
    }

    @Override
    public String toString(){
        return "EmpDetails [no=" + no + ", fname=" + fname + ", lname=" + lname + ", mail=" + mail + "]";
    }
}
```

Main.java.

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

     Session ses = HibernateUtil.getSession();
   Criteria ct = ses.createCriteria(com.s.h.m.EmpDetails.class);
     //execute logic
   List<EmpDetails> list = ct.list();
     //process the result
     for(EmpDetails eb:list){
              s.o.p(eb);
     }
   Criteria ct1 = ses.createCriteria(EmpDetails.class);
     //prepare conditions
   Criterion cond1 = Restrictions.between("no", 100, 900);
   Criterion cond2 = Restrictions.like("mail","%x.com");
     //add conditions
      ct1.add(cond1);
      ct1.add(cond2);
       //execute QBC logic
   List<EmpDetails> list = ct1.list();
        //process the result
     for(EmpDetails eb:list){
        s.o.p(eb);
        }
Criteria ct2 = ses.createCriteria(EmpDetails.class);
   Criterion cond1 = Restrictions.ge("no", 200);
   Criterion cond2 = Restrictions.le("no", 400);
   Criterion andcond = Restrictions.and(cond1, cond2);
   Criterion cond3 = Restrictions.like("mail","%x.com");
   Criterion orcond = Restrictions.or(andcond, cond3);
      ct2.add(orcond);
     Order o1 = Order.desc("no");
        ct2.addOrder(o1);
```

```
List<EmpDetails> list = ct2.list();
    for(EmpDetails eb:list){
            s.o.p(eb);
    }
Criteria ct3 = ses.createCriteria(EmpDetails.class);
Criterion cond1 = Restrictions.sqlRestriction("firstname like 'r%' ");
    ct3.add(cond1);
List<EmpDetails> list = ct3.list();
    for(EmpDetails eb:list){
        s.o.p(eb);
    }
//working with Projections to get specific multiple property values (col values)
    Criteria ct = ses.createCriteria(EmpDetails.class);
    Projection p1 = Projections.property("no");
    Projection p2 = Projections.property("mail");
    ProjectionList pl = Projections.projectionList();
        pl.add(p1); pl.add(p2);
    ct.setProjection(pl);
    List<Object[]> list = ct.list();
        for(Object row[]:list){
            s.o.p(row[0]+" "+row[1]);
        }
//using projections with Criteria API to work with aggregate operations
    Criteria ct4 = ses.createCriteria(EmpDetails.class);
//prepare multiple Projections objs for aggregate functions
    Projection p1 = Projections.count("no");
    Projection p2 = Projections.avg("no");
    Projection p3 = Projections.max("no");
    //add Multiple Projection objs to ProjectionList
    ProjectionList pl = Projections.projectionList();
    pl.add(p1); pl.add(p2); pl.add(p3);
    ct4.setProjection(pl);
    List<Object[]> list = ct4.list();
    Object res[] = (Object[]) list.get(0);
    s.o.p("records count"+res[0]);
    s.o.p("avg value of EID col"+res[1]);
}} s.o.p("max value of EID col"+res[2]); ses.close();
```

H10815

Native SQL

⇒ This is another Bulk operation technique to perform both select and non-select operations.

⇒ With this Native SQL, we can directly execute SQL commands on a database from hibernate application.

⇒ The two reasons given by hibernate for Native SQL.

1) For programmer convenience
2) To migrate a JDBC application as a hibernate application easily.

⇒ The one problem with Native SQL is hibernate application becomes database dependent.

⇒ To run SQL command, first we need to create SQLQuery object. We can create this object by calling createSQLQuery().

⇒ For select operation, we call list method & for non-select operation we call executeUpdate() method.

Ex1: SQLQuery qry = session.createSQLQuery("select * from emp");
List list = qry.list();

→ Here list contains Object[]'s, but not POJO class object. Because, sql command does not contains POJO classname.

Ex2:

SQLQuery qry = session.createSQLQuery("select * from emp");
qry.addEntity(Employee.class);
List list = qry.list();

→ Here list contains POJO objects, because we added entity name to qry object.

→ This type of qry object is called "entity qry" object.

Ex3)

SQLQuery qry = session.createSQLQuery("select empno, ename from emp");
List list = qry.list();

→ Here list contains Object[]'s.

→ Here Hibernate uses ResultSetMetaData to find the types of columns.

Ex(4)

```
SQLQuery qry = session.createSQLQuery("select empno, ename from emp");
        qry.addScalar("empno", Hibernate.INTEGER);
        qry.addScalar("ename", Hibernate.STRING);
    List list = qry.list();
```

→ here list contains object[]'s.

→ Hibernate does not use ResultSetMetaData, because we only specified column types

→ This type of query object is called "scalar query" object.

Ex(5):

```
SQLQuery qry = session.createSQLQuery("insert into emp values (?,?,?,?)");
        qry.setParameter(0,1234);
        qry.setParameter(1,"abcd");
        qry.setParameter(2,4000);
        qry.setParameter(3,20);
Transaction tx = session.beginTransaction();
    int i = qry.executeUpdate();
      tx.commit();
```

Program

Employee.hbm.xml

```
<hibernate-mapping>
 <class name="com.s.h.m.EmpDetails" table="Employee">
 <id name="no" column="EID"/> <!--Singular Id field cfg -->
 <property name="fname" column="FIRSTNAME"/>
 <property name="lname" column="LASTNAME"/>
 <property name="mail" column="EMAIL"/>
 </class>
<sql-query name="test1">
 <return class="com.s.h.m.EmpDetails"/>
   select * from Employee where email like :domain
</sql-query>
</hibernate-mapping>
```

EmpDetails.java

```
public class EmpDetails{

    private int no;
    private string fname, lname, mail;

  public EmpDetails(){
        S.O.P("EmpDetails :0-param constructor");
  }
  public EmpDetails(int no, string fname, string lname, string mail){
      S.O.P("EmpDetails : 4-param constructor");
        this.no = no;
        this.fname = fname;
        this.lname = lname;
        this.mail = mail;
      }
  //setters and getters

}
```

HibernateUtil.java

```
public class HibernateUtil{
    private static SessionFactory factory=null;
    static{
      try{
       Configuration cfg = new Configuration().configure("/com/nt/cfgs/hibernate.cfg.xml");
     ServiceRegistryBuilder builder=new ServiceRegistryBuilder();
     ServiceRegistry registry = builder.applySettings(cfg.getProperties()).buildServiceRegistry();
      }
    catch(Exception e){
          e.printStackTrace();
      }
}//static
```

```
static ThreadLocal <session> tl = new ThreadLocal <Session>();
    static Session ses =null;
    public static session getSession(){
        try{  if (tl.get()==null){
                                    ses=factory.openSession();
                                    tl.set(ses);
                                    return ses;
                                }
                else {
                                    return tl.get();
                                }
            }
        catch(Exception e){
            return null;
        }
    }//method
public static void closeSession(){
        try{
            ses.close();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }// closeSession()
}//class
```

Main.java

```
public class Main{
    public static void main(String[] args){
        Session ses =HibernateUtil.getSession();
    SQLQuery q1 =ses.createSQLQuery("select * from Employee where email like:
                                                                domain");
        q1.setString("domain","%x.com");
        q1.addEntity(EmpDetails.class);
      List <EmpDetails> list =q1.list();
        for(EmpDetails eb:list){
                S.O.P(eb);
        }
```

```
//non scalar Query
SQLQuery q2=ses.createSQLQuery("select EID,FIRSTNAME from Employee");
    q2.addScalar("EID",StandardBasicTypes.INTEGER);
    q2.addScalar("FIRSTNAME",StandardBasicTypes.STRING);

     List<Object[]> list = q2.list();
        for(Object[] row:list){
            for(Object o:row){
                    S.O.P(o+" ");
            }//for
        }//for
//Executing non-select Native SQL query to insert record
    Transaction tx=null;
    try{
        tx=ses.beginTransaction();
     SQLQuery q3=ses.createSQLQuery("insert into Employee values(:no,:fname,
                                        :lname,:mail)");
        q3.setInteger("no",1001);
        q3.setString("fname","Raja");
        q3.setString("lname","Rao");
        q3.setString("mail","x@y.com");
        int res=q3.executeUpdate();
        if(res==0)
           S.O.P("record not inserted");
        else
           S.O.P("record inserted");
     }
    catch(Exception e){
            tx.rollback();
     }
  //get Access to Named Native SQL Query
    Query q=ses.getNamedQuery("ntest1");
        q.setString("domain","%x.com");
     List<EmpDetails> list=q.list();
      for(EmpDetails eb:list){
       S.O.P(eb);
      }
   ses.close();
}//main
}
```

15/08/15

Named Queries

⇒ Named Queries are two types

1) Named HQL Query
2) Named Native SQL Query

⇒ If we want to run same query from multiple places of the project then we use concept of Named Queries.

⇒ With Named Queries, we can create the query for once and we can assign a name for it. We can run that query for multiple times with the help of its name.

⇒ We need to configure the query with some name in mapping file.

⇒ To configure HQL Query, we need to use <query> tag and to configure Native SQL Query we need to use <sql-query> tag.

⇒ for example,

employee.hbm.xml

```
<hibernate-mapping>
    <class ---  --->
    </class>
    <query name="q1"> from Employee e where e.deptNumber = ? </query>
    <sql-query name="q2"> select * from emp where deptno = ? </sql-query>
</hibernate-mapping>
```

⇒ We can run a Named Query like the following,

```
Query qry = session.getNamedQuery("q1");
qry.setParameter(0,20);
List list = qry.list();

Query qry2 = session.getNamedQuery("q2");
qry2.setParameter(0,20);
List list = qry2.list();
```

Hibernate relationships

Why relationships in Database?

⇒ We can store our appln data in a database table, to make it as a persistent data.

⇒ We have an appln which contains doctors & patients data, we can store that data in a single table as like,

doctorId	DocName	Qlf	patientId	patientName	Addr
701	Mr.A	MD	1001	X	Hyd
701	Mr.A	MD	1002	Y	Hyd
701	Mr.A	MD	1003	Z	Hyd
702	Mr.B	MS	9001	P	Hyd
702	Mr.B	MS	9002	Q	Hyd
702	Mr.B	MS	9003	R	Hyd

In the above table, there is a data redundancy problem. To reduce the data redundency, we need to divide the data and we need to store in a 2 different tables.

⇒ In order to get the relation between the data, we also need to take the column of one table as a common in another table.

doctor (parent)

↓PK doctorId	docName	Qlf
701	Mr.A	MD
702	Mr.B	MS

patient (child)

↓PK patientId	patientName	Addr	↓FK docId
1001	X	Hyd	701
1002	Y	Hyd	701
1003	Z	Hyd	701
9001	P	Hyd	702
9002	Q	Hyd	702
9003	R	Hyd	702

⇒ By dividing the data as above tables, only a single column is going to have duplicate values. We reduced the data redundancy by adding the relationship

17/08/15

⇒ In hibernate appln, if we create single POJO class and if we set the data to that object then there is a chance of getting data redundency.

⇒ Data redundency means, some data in multiple objects of a POJO class can be duplicated.

⇒ In order to reduce the redundency, we divide the properties of one class in to two classes and then we apply a relationship bit objects of the two classes.

⇒ In hibernate, we can apply 4 types of relationships bit POJO classes,

1) one-to-many
2) many-to-one
3) many-to-many
4) one-to-one

1) One-to-Many

⇒ In hibernate, if we want to apply one-to-many relation from a parent object to multiple child objects then first we need to add child objects to collection and then we need to set that collection object to parent.

⇒ In order to set a collection object, in parent class we need to create a reference variable of type collection.

⇒ We can use collection type as Set (or) List (or) Map.

⇒ For example, we have a Customer (parent class) and Item (child class). When creating Customer class, we need to create reference variable of type collection.

```
//parent
public class Customer
{
  private int customerId;
  private String customerName;
  private Set items;
  //setters and getters
}
```

```
// child
public class Items
{
  private int itemId;
  private String itemName;
  private double price;
  // setters and getters
}
```

⇒ To add one-to-many relationship from a Customer to three Items the code will be like the following,

```
Set items = new HashSet();
items.add(i1);
items.add(i2);
```

```
items.add(i3);
c1.setItems(items);
```

customer.hbm.xml

```
<hibernate-mapping>
  <class name="Customer" table="customer">
   <id name="customerId" column="custid" type="int" length="3" />
    <property name="customerName" column="custname" type="string" length="10" />
    <set name="items">
      <key column="custid_fk" />
      <one-to-many class="Items" />
    </set>
  </class>
</hibernate-mapping>
```

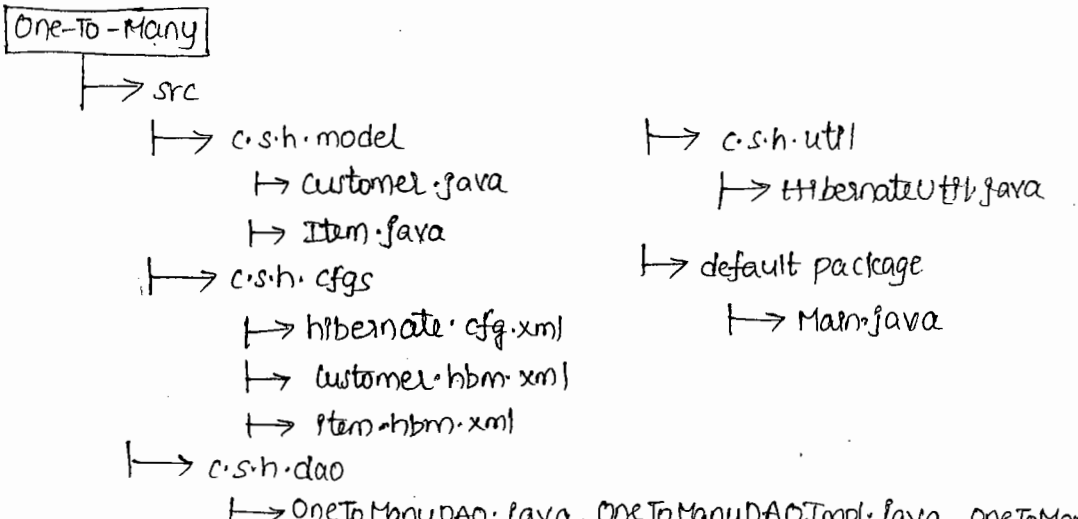
⇒ By default, hibernate will only save parent class object but not it's child class objects one-to-many relationship.

⇒ In order to save child class objects also in database along with parent class object, we need to set cascade attribute value as all in <set> tag.

```
<set name="items"
     cascade="all">
<key column="custid_fk" />
<one-to-many class="Items" />
</set>
```

18/08/15

Project Directory structure



Customer.java

```
public class Customer{

    private int customerId;
    private String customerName;
    private Set items;

    //setters & getters

    public int getCustomerId(){
        return customerId;
    }

    public void setCustomerId(int customerId){
        this.customerId = customerId;
    }

    public String getCustomerName(){
        return customerName;
    }

    public void setCustomerName(String customerN
    { this.customerName = customerName;
    }

    public Set getItems(){
        return items;
    }

    public void setItems(Set items){
        this.items = items;
    }

}

public class Item {
    private int itemId;
    private String itemName;
    private int price;

    //setters & getters

    public int getItemId(){
        return itemId; }

    public void setItemId(int itemId){
        this.itemId = itemId; }

    public String getItemName(){
        return itemName; }

    public void setItemName(String itemName){
        this.itemName = itemName; }

    public int getPrice(){
        return price; }

    public void setPrice(int price){
        this.price = price; }

}
```

customer.hbm.xml

```
<hibernate-mapping>
  <class name="com.sathya.hibernate.model.Customer">
  <id name="customerId" column="custid" />
  <property name="customerName" column="custname" length="10" />
  <set name="items" cascade="all">
    <key column="custid-fk" />
   <one-to-many class="com.sathya.hibernate.model.Item" />
  </set>
 </class>
</hibernate-mapping>
```

Item.hbm.xml

```
<hibernate-mapping>
<class name="com.sathya.hibernate.model.Item">
<id name="ItemId" column="ItemId" />
 <property name="ItemName" column="ItemName" length="10" />
 <property name="price" />
</class>
</hibernate-mapping>
```

hibernate.cfg.xml

```
<mapping resource="com/sathya/hibernate/cfgs/customer.hbm.xml" />
<mapping resource="com/sathya/hibernate/cfgs/Item.hbm.xml" />
```

OneToManyDAO.java

```
Public interface OneToManyDAO{

    void saveCustomer();

}
```

```
public class OneToManyDAOImpl implements OneToManyDAO {

    @Override
    public void saveCustomer() {

        Customer c1 = new Customer();

            c1.setCustomerId(101);
            c1.setCustomerName("Rafa");

        Item i1 = new Item();

            i1.setItemId(9001);
            i1.setItemName("iphone");
            i1.setPrice(10000);

        Item i2 = new Item();

            i2.setItemId(9002);
            i2.setItemName("samsung");
            i2.setPrice(9000);

        Set items = new HashSet();

            items.add(i1);
            items.add(i2);

        c1.setItems(items);

    SessionFactory factory = Hibernateutil.getSessionFactory();

        Session ses = factory.openSession();
    Transaction tx = ses.beginTransaction();
            ses.save(c1);
            tx.commit();
            ses.close();
    }

}
```

abstract Design Pattern

```
public class OneToManyDAOFactory {

    public static OneToManyDAO getInstance()
    {
        return new OneToManyDAOImpl();
    }
}
```

Main.java

```
public class Main {

    public static void main (String [] args)
    {
        OneToManyDAO dao = OneToManyDAOfactory.getInstance();
            dao.saveCustomer();
    }
}
```

dp > select * from Customer;

CUSTID	CUSTNAME
101	Raja

>select * from item;

ITEMID	ITEMNAME	PRICE	CUSTID-FK
9001	Iphone	10000	101
9002	samsung	9000	101

How to add onemore child object to existing parent

⇒ If we want to add one or more new child objects to existing parent then we need to add the new child objects to the existing collection.

⇒ If we want to add one more item to the customer then the following code is needed.

```
Customer c1 = (Customer) session.get (Customer.class, 101);
    Set items = c1.getItems();
    Transaction tx = session.beginTransaction();
        items.add(i3);
        tx.commit();

    Item i3 = new Item();
        i3.setItemId(9003);
        i3.setItemName("Moto");
        i3.setPrice(7000);
```

19/08/15

⇒ In one-to-many example, to implement this operation, do the following changes,

1) Open DAO Interface, add the following method signature.

```
void addAnotherItem();
```

2) Open DAOImpl class and define the method like the following.

```
public void addAnotherItem()
{
    Item i2 = new Item();
        i2.setItemId(9003);
        i2.setItemName("moto");
        i2.setPrice(7000);
    SessionFactory factory = HibernateUtil.getSessionFactory();
      Session session = factory.openSession();
      Customer c1 = (Customer) session.get(Customer.class, 101);
        Set items = c1.getItems();
    Transaction tx = session.beginTransaction();
          items.add(i2);
          tx.commit();
        session.close();
}
```

How to remove a child object from parent?

⇒ To remove one child object from a parent, ie, to remove an item from customer, we need to follow the below steps.

1) Read the parent (customer) from database.
2) Read the collection from customer.
3) Read an item from database, which we want to remove from parent.
4) Remove item from collection, with in a transaction.

Ex:

```
Customer c1 = (Customer) session.get(Customer.class, 101);
    Set items = c1.getItems();
Item i3 = (Item) session.get(Item.class, 9003);
Transaction tx = session.beginTransaction();
        items.remove(i3);
    tx.commit();
```

⇒ Hibernate updates foreign key as "null", to cut the relationship bit a parent record and child record, but it will not delete the child record.

⇒ For a record, if foreign key is "null" then it is called an "orphan record".

⇒ Item

ItemID	ItemName	Price	custid-fk
9001	iphone	10000	101
9002	samsung	9000	101
9003	Moto	7000	null → orphan record.

⇒ To tell the hibernate about delete an orphan record immediately from table, we need to change the cascade attribute value as "all-delete-orphan"

⇒ The possible values of cascade attribute are,

1) none (default) apply all operations on parent only.
2) all (apply all operations on child & its parent)
3) all-delete-orphan (apply all operations.
4) save-update (apply insert & update operations of parent on its child)
5) delete (delete operation of parent on its child)

⇒ If we set cascade = "save-update" then hibernate applies insert and update operations of parent on its child.

⇒ If we set cascade = "delete" then hibernate applies only delete operation of parent on its child.

⇒ If we set cascade = "all" then hibernate applies insert, update & delete operations of parent on its child.

20/8/15 Lazy attribute

⇒ In one-to-many relationship when we are mapping in the collection in hbm file, we can add "lazy" attribute to a <collection-mapping> tag.

⇒ The default value of lazy attribute is "true".

⇒ If lazy="true" then hibernate will not load a collection with a parent it means only parent is selected without its child we call it as lazy loading.

⇒ If lazy="false" then hibernate loads a collection also with parent, it means a parent is loaded with its child we call it as Easly loading.

⇒ If we want only parent information and with parent its childs are also loaded unnecessarly from database then there is a perfomance drawback. In this case we apply lazy loading, Hibernate internally creates a proxy collection object and it will set that proxy object to parent object.

⇒ We can identify whether with a parent its collection is also loaded or not by using select operations display on the console.

Deleting a customer (parent)

⇒ In one-to-many associations, when a parent object is deleted then automatically its child objects are also deleted by hibernate.

⇒ First hibernate deletes, child objects and then deletes parent obj.

Ex: Customer c1 = (Customer) session.get(Customer.class, 101);

```
Transaction tx = session.beginTransaction();
        session.delete(c1);
        tx.commit();
```

Changing collection type as List:

⇒ In one-to-many relationship, in a parent class we need a reference variable of collection type and that collection-type can be Set (or) List (or) Map.

⇒ If collectiontype is List then in mapping file (hbm) we need to configure that collection using <list> tag.

⇒ when saving child objects in database hibernate will save the index of the child also in table so in child table one extra column is needed is List index column.

Ex public class customer {

```
private int customerId;
private String customerName;
private List item;

//setter and getter
```

⇒ In hbm file we need to map the list using <list> tag like the following,

```
<list name="items" cascade="all">
<key column="custid-fk"/>
<list-index column="li"/>
<one-to-many class="Item."/>
</list>
```

(li → extra column)

⇒ We can add child objects to parent when collection type is List is like the following,

```
List items = new ArrayList(); // List type is always integer.
items.add(i1);
items.add(i2);
c1.setItems(items);
```

21/8/15

⇒ In one-to-many association, if we use collection type as List in parent class then in hbm file we can map that collection by using either <list> (or) <bag> (or) <idbag>

⇒ <idbag> tag is only used in a Many-To-Many relationship.

⇒ In one-to-many relationship, we can use either <list> tag (or) <bag> tag.

⇒ If we want to insert child records of parent record along with there index then we use <list> tag.

⇒ If we want to store child records of a parent record, with out index then we use <bag> tag.

⇒ for example in customer.hbm.xml, we can configure a <bag> tag like the below

customer.hbm.xml

```
<hibernate-mapping>
<class name="c.s.h.m.Customer" table="Customer">
  ---
  ---
  <bag name="items" cascade="all">
   <key column="cust_id_fk" />
   <one-to-many class="com.s.h.m.Item" />
   </bag>
  </class>
</hibernate-mapping>
```

java.util.Set	→ <set>
java.util.List	→ <list> | <bag> | <idbag>
java.util.Map	→ <map>

Changing collection type as Map

⇒ In one-to-many association, we need a reference variable of collection in parent class.

⇒ If we choose collection type as Map then .hbm file we need to configure <map> tag.

⇒ When hibernate is inserting a child records it will also insert its key used in a map.

⇒ In child table of DB one extra column is required for storing map keys we call it as a map key column.

⇒ In case of collection type as map, it is not possible to avoid map key column from child table.

⇒ To use collection-type as map, we need to do following changes in one-to-many application.

"Customer.java

```
public class Customer
{
    private int customerId;
            String customerName;
            Map items;
    // setters and getters
    }
```

customer.hbm.xml

```
<hibernate-mapping>
 <class name = "Customer" table ="customer">
    <id name = "customerId" column ="custid"/>
    <property name = "customerName" column ="custname" type ="string" />
  <map name ="items" cascade ="all">
    <key column ="custid-fk" />
    <map-key column ="mkey" type ="string" />
    <one-to-map class ="item" />
  </map>
  </hibernate>
 </class>
</hibernate-mapping>
```

⇒ We can add two items as childs to customer like the following,

```
Map items = new HashMap();
    items.put ("k1", i1);
    items.put ("k2", i2);
  c1. setItems (items);
```

22/8/15 Many-To-One

⇒ Many-To-one association is used to add relationship from child class object to its parent class object.

⇒ If relationship is added from child class object to parent class object then operations done on child class object will be effected on its parent class object.

⇒ In many-to-one association, we need to set parent class object to its child objects so in child class we need to create parent class reference variables.

⇒ In hibernate when the relationship to-many type then we need a collection type reference variable in parent class and if relationship in to-one then we need a parent class reference variable in child class.

For ex

⇒ If we want to apply Many-to-one relationship from items to customer then we need customer class reference variable in item class.

```
public class Item {
    private int itemId;
            string itemName;
            double price;
    private Customer customer;
    //setters and getters
}
```

⇒ In hbm file, to tell the hibernate that the relationship is Many-To-one, we need to configure <Many-To-one> tag.

items.hbm.xml

```
<hibernate-mapping>
<class name="c.s.h.m.Item" table="items">
    <id name=
    <property name=
    <property name=
<many-to-one class="c.s.h.m.Customer" name="customer" cascade="all"
                                    column="custId-fk" />
</class>
</hibernate-mapping>
```

⇒ When we save a child object then hibernate will save ~~only child~~ objects. its parent object also in DB.

⇒ If parent is already exists hibernate will save only child objects.

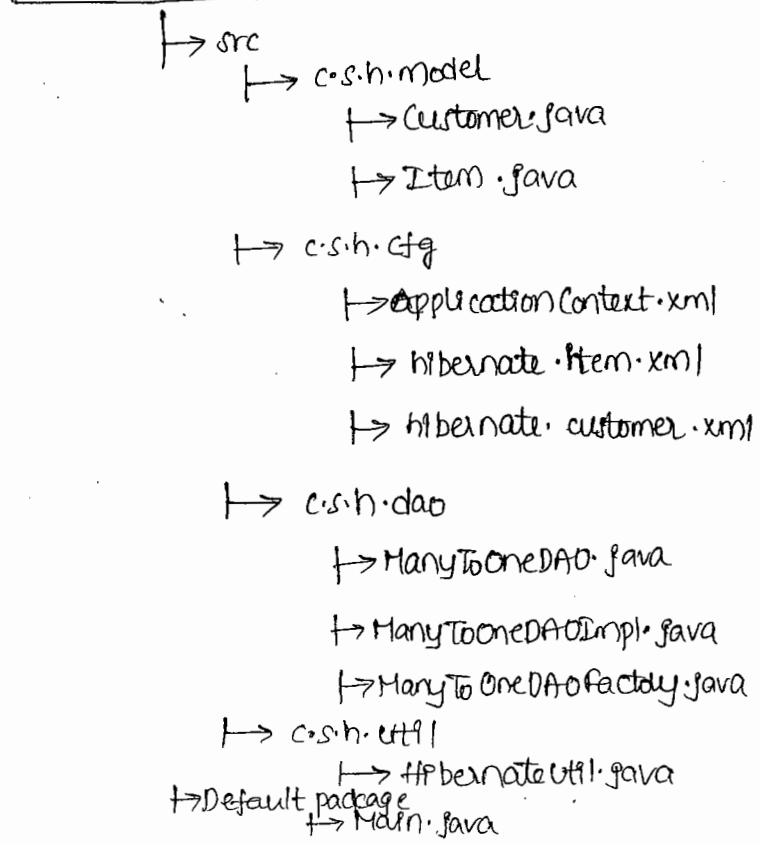
```
I1.setCustomer(c1);
I2.setCustomer(c1);

Transaction tx = session.beginTransaction();
        session.save(I1);
        session.save(I2);
       tx.commit();
```

⇒ When we are deleting child object hibernate will delete its parent object also if parent has more childs then exception will be thrown.

⇒ In order to delete only child object without parent, we need to change cascade attribute value to "none" (or) save-update.

```
Item I = (Item) session.get(Item.class, 9002);
Transaction tx = session.beginTransaction();
    session.delete(I);
   tx.commit();
```

Many-To-One



Customer.java

```
public class Customer {
    private int customerId;
    private String customerName;
    //setters and getters
}
```

Item.java

```
public class Items {
    private int itemId;
        String itemName;
        int price;
    private Customer customer;
    //setters and getters
}
```

customer.hbm.xml

```
<hibernate-mapping>
<class name="c.s.h.model.Customer" table="customer">
  <id name="customerId" column="customerId"/>
  <property name="customerName" column="customername"/>
  </class>
</hibernate-mapping>
```

Item.hbm.xml

```
<hibernate-mapping>
   <class name=
   <id
   <property
   <property
   <many-to-one class="c.s.h.model.Customer" name="customer" cascade="all"
                column="cust-id-fk"/>
   </class>
</hibernate-mapping>
```

ManyToOneDAO.java.

```
public interface ManyToOneDAO {
        void saveItem();
        void deleteItem();
    }
```

ManyToOneDAOFactory.java

```
public class ManyToOneDAOFactory {
    Public static ManyToOneDAOFactory getObject(){
} }     return newManyToOneDAOImpl();
```

Many-To-one DAOImpl.java.

```
public class ManyToOneDAOImpl implements ManyToOneDAO
{   @Override
    public void saveItem()
    {   Customer c1 = new Customer();
        c1.setCustomerId (101)
        c1.set CustomerName ("A");
        Item i1 = new Item();
        i1.setItemId(9001);
        i1.setItemName("x");
        i1.setPrice(5000);
        Item i2 = new Item();
        i2.setItemId(9002);
        i2.setItemName("Y");
        i2.setPrice(9000);
        //set Parent object to childs
        i1.setCustomer(c1);
        i2.setCustomer(c1);
    SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();
            session.save(i1);
            session.save(i2);
            tx.commit();
            session.close();
    }
    @Override
    public void deleteItem()
    {
        Item i = (Item) ses.get (Item.class, 9002);
        Transaction = tx = ses.beginTransaction();
            ses.delete(i);
            tx.commit();
            ses.close();
    }
}
```

Main.java.

```
public class Main { p.s.v.m(String[] args) {
    ManyToOneDAO dao = ManyToOneDAOfactory.getObject();
    } } dao.saveItem();
```

24/8/15

Lazy Attribute

⇒ In Many-to-One relationship, default value of "lazy" attribute is "proxy".

⇒ If lazy = "proxy" then, with child its parent will not be loaded. Instead a proxy instance is created for parent and that proxy object will be set to the child object.

⇒ If lazy = "false" then, along with child object its parent object is also loaded.

⇒ If lazy = "no-proxy" then, with child its parent is not loaded, but an empty parent object will be created and it will be set to the child object.

⇒ The output for lazy = "proxy" and lazy = "noproxy", for both it is same, but internally, for "proxy" a proxy object created and for "no-proxy" empty object created.

One-to-Many Bidirectional

⇒ If we apply only one-to-many relationship (or) only many-to-one relationship then uni-directional association.

⇒ In uni-directional association, we can perform operations either only from parent to effect on child (or) only from child to effect on parent.

⇒ In an application, sometimes we want to perform operation from parent and sometimes from child, then we need to apply bi-directional one-to-many association.

⇒ To apply bi-directional relationship, we need to combine both one-to-many and many-to-one relationships.

⇒ For one-to-many, we need to create a reference variable of collection type in parent class and parent class reference in child class for many-to-one.

```
public class Customer
{
  private set items;
}
```

```
public class Item
{
  private Customer customer;
}
```

customer.hbm.xml

```
<set name = "items" cascade = "all">
  <key column = "custId_fk" />
<one-to-many class = "Item" />
</set>
```

item.hbm.xml

```
<many-to-one class = "Customer"
             name = "customer"
             cascade = "all"
             column = "custId_fk"
             lazy = "proxy" />
```

⇒To get bi-directional association, first we need to set parent object to the child objects and add all child objects to the collection and we need to set collection to the parent object.

```
p1.setCustomer(c1);
p2.setCustomer(c2);
Set items = new HashSet();
items.add(p1);
items.add(p2);
c1.setItems(items);
```

⇒Suppose if we save(p1) i.e., session.save(p1) then with p1 its parent c1 is saved c1 is parent, so when it is saved its child's also saved. p2 is automatically saved

2 Steps

Many-to-Many

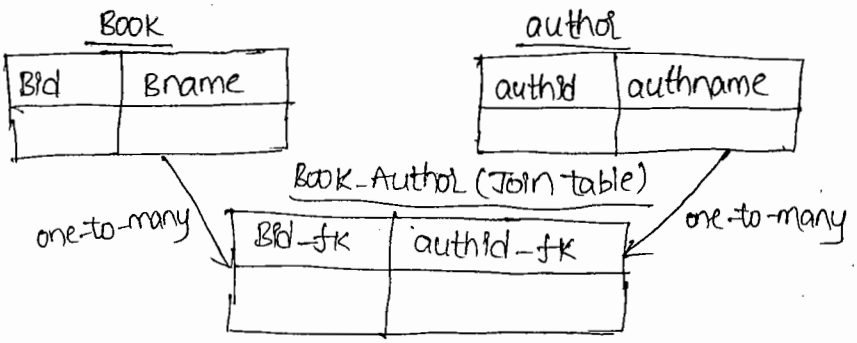
⇒Many-to-Many is the relationship, which is a combination of one-to-many from first POJO class to second POJO class and it is again one-to-many from second POJO class to first POJO class.

⇒For example, if we have two classes Book & Author then one-to-many relationship from book to author and again one-to-many relationship from author to book is combinely called a "Many-to-Many" relation.

⇒In database, only with two tables, we cannot apply many-to-many relationship. We need a third table called a "Join table" also.

⇒In database, we need to apply one-to-many relationship from table1 to Jointable and from table2 to Jointable, to get many-to-many relationship between table1 and table2.

⇒If the relationship is many-to-many between book & author then in the database we need tables like the below,



⇒In hibernate, when many-to-many relationship required between a book & author POJO classes. Then in both POJO classes, we need a reference variable of type collection.

```
public class Book
{
  private int bookId;
  private String bookName;
 private Set authors;
  //setters & getters
  --
}
```

```
public class Author
{
 private int authorId;
 private String authName;
 private List books;
 //setters & getters
 --
}
```

book.hbm.xml

```
<hibernate-mapping>
  <class name="Book">
    <id name="bookId" column="bid" />
    <property name="bookName" column="bname" length="8" />
    <set name="authors" cascade="all" table="Book_Author">
                                          → Join table
      <key column="bid_fk" />
            → foreign key
    <many-to-many class="Author" column="authId_fk" />
                                  → inverse foreign key
    </set>
  </class>
</hibernate-mapping>
```

author.hbm.xml

```
<hibernate-mapping>
  <class name="Author">
    <id name="authorID" column="authid" />
    <propety name="authorName" column="authname" length="10" />
    <bag name="books" cascade="all" table="Book_Author">
      <key column="authid_fk" />
          → foreign key
      <many-to-many class="Book" column="bid_fk" />
                                  → inverse foreign key
    </bag>
  </class>
</hibernate-mapping>
```

Many-To-Many

- src
  - com.sathya.hibernate.model
    - Book.java
    - Author.java
  - com.sathya.hibernate.cfgs
    - book.hbm.xml
    - author.hbm.xml
    - hibernate.cfgs.xml
  - com.sathya.hibernate.dao
    - ManyToManyDAO.java
    - ManyToManyImpl.java
    - ManyToManyDAOfactory.java
  - com.sathya.hibernate.util
    - HibernateUtil.java
  - default
    - Main.java

26/08/15

Book.java

```
public class Book{
    private int bookId;
    private string bookName;
    private set authors;

    // setters and getters

    }
```

Author.java

```
public class Author{

    private int authorId;
    private String authorName;
    private List books;

    // setters and getters

    }
```

book.hbm.xml

```
<hibernate-mapping>
 <class
 <id name="bookId" column="bid" />
 <property name="bookName" column="bname" />
 <set name="author" cascade="all" table="book-author" />
 <key column="bid-fk" />
 <many-to-many class="pkg.Author" column="authid-fk" />
 <set>
</class>
</hibernate-mapping>
```

↑ inverse foreign key

author.hbm.xml

```
<hibernate-mapping>
<class name="
 <id name="authId" column="authId" />
<property name="authorName" column="authName" />
<proper bag name="books" cascade="all" table="Book-Author" />
 <key column="authid-fk" />
<many-to-many class="pkg.model.Book" column="bid-fk" />
 <bag>
 <id>
</class>
</hibernate-mapping>
```

ManyToManyDAO.java

```
public interface ManyToManyDAO{
    void saveBooks();
}
```

ManyToManyDAOImpl.java

```
public class ManyToManyDAOImpl{
    @Override
    public void saveBooks(){
        Book b1 = new Book();
        b1.setBookId(1);
        b1.setBookName("Java");

        Author a1 = new Author();
        a1.setName("A");
        a1.setId("20");

        Author a2 = new Author();
        a2.setName("B");
        a2.setId("30");

        Set author1 = new HashSet();
        author1.add(a1);
        author1.add(a3);

        Book b2 = new Book();
        b2.setBookId(2);
        b2.setBookName("Oracle");

        Author a3 = new Author();
        a3.setId("40");
        a3.setName("[redacted]");

        Set author2 = new HashSet();
        author2.add(a1);
        author2.add(a2);

        b1.setAuthor(author1);
        b2.setAuthor(author2);

        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();
        session.save(b1);
        session.save(b2);
        tx.commit();
        session.close();
    }
}
```

Main.java

```
public class Main{
    public static void main(String[] args){
        ManyToManyDAO dao = ManyToManyDAOFactory.getObject();
        dao.saveBooks();
    }
}
```

26/8/15

SQL> select * from book;

BID	BNAME
1	java
2	oracle

SQL> select * from author;

AUTHID	AUTH NAME
101	A
102	B
103	C

SQL> select * from book_author;

AUTHID_fk	BID_fk
101	1
103	1
101	2
102	2

<idbag> tag:

⇒ In Many-To-Many relationship, join table contains foreign_keys. Foreign keys can have duplicate values. so there is no uniqueness column. If there is a unique column like a primary key column in join table then it will be easy to perform directly operations on join table column by using sql commands.

⇒ If a primary key column is added to the join table then to tell the hibernate that primary key column of join table, we use <idbag> tag in "hbm" file.

⇒ In the above Many-To-Many, if we want to configure <idbag> tag then we need to do the following changes,

1) In Book class change collection type as List.
2) In book.hbm.xml we need to configure <idbag> tag.

```
<idbag name="authors" cascade="all" table="book_author">
  <collection-id column="bid_authid_pk" type="int">
      <generator class="increment" />
  </collection-id>
  <key column="bid_fk" />
  <many-to-many class="pkg.Author" column="authid_fk" />
</idbag>
```

⇒ 3) In DAOImpl class, instead of creating HashSet class object, we need to create ArrayList class object.

- 4) After executing main class, we will get the data in JOIN table like below,

book-author

BID_FK	AUTHID_FK	BID_AUTHID_PK
1	101	1
1	103	2
2	101	3
2	102	4

2-10-15

One-To-One Relationship

⇒ One-To-One relationship can be applied in 2 ways.

1) One-To-One with foreign key
2) One-To-One with Primary key

One-To-One with Foreign key (FK)

⇒ It is similar to many-To-one relationship only. But foreign key does not allows duplicate and null values.

⇒ In mapping file, <many-to-one> tag with 2 new attributes unique="true" and not-null="false".

⇒ In the following example, we are creating two POJO classes Person & Passport. The relationship is one passport to one person.

Here passport is a child class & person is a parent class.

⇒ In hibernate, to apply a one-to-one relationship we need a parent class reference variable in child class.

⇒ The two tables are needed in DB are :-

Person

Pid	pname

Passport

pno	expdate

One-To-One

```
|→ src
    |→ c.s.h.m
        |→ Person.java
        |→ Passport.java
    |→ c.s.h.c
        |→ person.hbm.xml
        |→ passport.hbm.xml
        |→ hibernate.cfg.xml
    |→ c.s.h.d
        |→ OneToOneDAO.java
        |→ OneToOneDAOImpl.java
        |→ OneToOneDAOfactory.java
    |→ c.s.h.u
        |→ HibernateUtil.java
    |→ default
        |→ Main.java
```

Person.java

```
public class Person
{
    private int personId;
    private String personName;
    //setters and getters
}
```

Passport.java

```
public class Passport{
    private int passportId;
    private Date expiryDate;
    private Person person;
    //setters and getters
}
```

person.hbm.xml

```
<hibernate-mapping>
<class name="com.s.h.m.Person">
<id name="personId" column="pid" />
<property name="personName" column="pname" type="string" length="8" />
</class>
</hibernate-mapping>
```

passport.hbm.xml

```
<hibernate-mapping>
<class name="
<id name="passportId" column="pno" />
<property name="expiryDate" column="expdate" type="date" />
<one-to-one class="com.s.h.Person" name="person" cascade="all" unique="pid-fk"
                                                notnull="false" />
</class>
</hibernate-mapping>
```

OneToOneDAO.java

```
public interface OneToOneDAO{
        void savePasspot();
    }
```

OneToOneDAOImpl.java

```
public class OneToOneDAOImpl implements OneToOneDAO{

    @Override
    public void savePassport(){

        Person person = new Person();
          person.setPersonId(1010);
          person.setPersonName("A");

        Passport passport = new Passport();
          passport.setPassportId(9080);

    // calender is abstract class & Gregorian calender is subclass.
    Here getInstance() is a static factory method, returns Gregorian
    Calender object.

        Calender calender = Calender.getInstance();
                calender.set(2019, 10,20);
        java.util.Date date = calender.getTime();
          passport.setExpiryDate(date); // expire date is 20/oct/2019.
            passport.setPerson(person);
    SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
    Transaction tx = session.beginTransaction();
            session.save(passport);
              tx.commit();
            session.close();

        }

}
```

OneToOneDAOfactory.java

```
public class OneToOneDAOfactory{

    public static OneToOneDAO getInstance()
    {
        return new OneToOneDAOImpl();
    }
}
```

Main.java

```
public class Main{

    public static void main(String[] args)
    {
        OneToOneDAO dao = OneToOneDAOfactory.getInstance();
            dao.savePasspolt();
    }
}
```

SQL> select * from person;

PID	PNAME
1010	A

SQL> select * from passpolt;

PNO	EXPDATE	PID_FK
9080	20-OCT-19	1010

One-to-one with Primary key

one-to-one with primary key is applicable, if in child table a primary key column is also acting as a foreign key column.

⇒ In one-to-one with primary key, parent record primary key & child record primary key, both values are always same.

⇒ In order to tell the hibernate that primary key column of child table is foreign key also, we need to configure generator class as "foreign" for id.

⇒ This foreign generator of hibernate is only applicable in one-to-one with primary key relationship.

⇒ In person & passport relationship, if we take primary key & foreign key in passport table as "pno" column then we need to configure foreign generator like the following in passport.hbm.xml

passport.hbm.xml

```
<hibernate-mapping>
 <class   name ="
 <id name=" com.s.h.m.
 <id  name ="passportId "  column="pno" />
 <generator  class =" foreign" >
 <param  name ="property " > person </param>
 </generator >
 </id>
 <property
 < one-to-one  class =" com.s.h.m. Person"  name ="person" cascade ="all " />
 </class>
 </hibernate-mapping>
```

28/08/15

Q What is (1+n) selects problem in hibernate ?

A Suppose we have an one-to-many relationship from customer to item.

⇒ There are 3 customers & each customer has some childs (items).

⇒ When we selects all the customers from table then hibernate executes one select operation for reading all the customers & then 3 select operations for reading childs of 3 parents (items of 3 customers)

⇒ Totally (1+3) selects are executed. Here

1 → indicates one select for all parents

3 → indicates 3 selects for reading childs of 3 parents.

⇒ If there are 'n' parents then (1+n) selects are generated. This is called (1+n) selects problem.

⇒ (1+n) selects will increase the no. of trips to database & decreases the perfamance of application.

Fetching strategies

⇒ Fetching strategies are used to overcome (1+n) selects problem.

⇒ There are 3 fetching strategies:

1) select (default)
2) Join
3) subselect

⇒ We can add a fetching strategies by adding fetch attribute to collection mapping tag.

⇒ <set name="items" cascade="all" lazy="false" fetch="select">

⇒ If fetch="select", then hibernate generates (1+n) selects.

⇒ If fetch="join", then hibernate selects all parents & these respective childs in a single select operation. It means (1+n) selects are reduced to a single select.

⇒ When we write fetch="join", then HQL again generate (1+n) selects. It means fetch="join" is not supported with "HQL", but supported with "criteria".

⇒ If fetch="subselect" then hibernate reads all parents with one select operation & reads all childs of parents with one more select operation, totally (1+n) selects are executed

Q When LazyInitializationException occurs?

A Suppose we loaded an object lazily, then we closed session and after that we are reading the data from lazily loaded object.

⇒ Because of session is already closed, connection is also already closed. so hibernate cannot read the data from database. Hence it throws LazyInitialization Exception.

For ex:

```
Product p = (Product) session.load(Product.class, 101);
session.close();
double price = p.getPrice();
```

⇒ In the above code, we called load method, so it loads a product lazily.

⇒ We are reading the price after closing the session so hibernate throws LazyInitializationException.

29/8/15

What is inverse attribute ?

⇒ In a one-to-many relationship, when we save a parent object, along with parent hibernate will save its child object and also hibernate updates the foreign key by generating separate update commands.

⇒ If a parent has "10" child objects then hibernate generates ten insert commands and 10 update commands to update the foreign key. These updates will decrease the performance.

⇒ If we tell the hibernate that owner of the relation has a child object then hibernate will not generate separate update commands to set the foreign key. So to tell hibernate that owner of the relationship is changed to child, we need to add inverse = "true" in collection-mapping tag.

HQL Join

⇒ In hibernate, to construct a join statement, we need relationship between the two pojo classes.

⇒ The purpose of writing a join statement is, with a single select query, we can read the data from multiple tables of database.

⇒ In hibernate, we have 4 types of joins.

1) inner join (default)
2) left join
3) right join
4) full join

⇒ By default a join type is inner join. It reads only related data from both sides of the join statement.

⇒ While construction a join statement, at the opposite side of the join, we need to use the reference variable used in a POJO class to join the relation.

⇒ For example,

select c.customerName, i.itemName from Customer c join c.items i.

⇒ The above join statement reads customerName & itemName using one-to-many relationship. Here, after the "join" items is reference variable of collection type.

⇒ select c.customerName, i.itemName from item i join i.customer c.

⇒ The above join statement reads customerName & itemName using Many-to-one relationship.

⇒ If we use Left join then the query reads related data from both sides and also unrelated data from left side of the join if any.

⇒ In case of right join, a query reads related data and unrelated data from right side if any.

⇒ In case of full join, hibernate reads related data and unrelated data from both side of the join statement.

```
Query query = ses.createQuery("select c.customerName, i.itemName from
                                customer c join c.items i ");
List list = query.list();
Iterator it = list.iterator();
    while(it.hayNext())
    {
      Object[] row = (object[]) it.next();
      S.o.p(row[0]+" "+row[1]);
    }
ses.close();
```

	customer	itemName
	row[0]	row[1]
	row[0]	row[1]
	row[0]	row[1]

Left Join

select c.customerName, i.itemName from item i left join i.customer c.

Right Join

select c.customerName, i.itemName from item i right join i.customer c.

Full Join

select c.customerName, i.itemNam from item i full join i.customer c.

31/08/15

Hibernate connection pooling

⇒ By default hibernate comes with built-in connection pool.

⇒ The built-in connection pool of hibernate is not advisable to use in a production level application. Because there is a lack of several features.

⇒ So, hibernate is recomanded to use a standalone third party tool for standalone applications and a sewer connection pool for sewer-side applications.

⇒ Hibernate itself distributed either to third party connection pools called "C3P0 & proxool."

⇒ If we want to use either C3P0 (or) proxool then we need to configure its related properties in hibernate.cfg.xml.

⇒ Mostly, we develop sewer-side appln's only for the real time. so in hibernate we use sewer-side connection pool.

⇒ In case of standalone applications, mostly we use C3P0 connection pool.

⇒ Suppose if we want to add C3P0 connection pool support then we need to configure the following 3 C3P0 properties additionally in hibernate.cfg.xml

```
<property name="C3P0.min-size"> 3 </property>
<property name="C3P0.max-size"> 20 </property>
<property name="connection.provider_class"> org.hibernate.connection.C3P0ConnectionProvider </property>
```

⇒ When C3P0 properties are added then we need to add C3P0.jar to the build path.

⇒ ~~If~~ In hibernate if we want to add Hibernate with sewer connection pool

⇒ If any appln wants to get a connection from sewer connection pool then the mediator object called DataSource is required.

⇒ In sewer, datasource object reference will be stored in a JNDI registry.

⇒ To connect with JNDI registry for reading the DataSource object reference, jndi properties are required.

⇒ In hibernate cfg file, we need to replace connection properties with jndi properties.

⇒ By using jndi properties, hibernate connects with registry, reads DataSource object referene

and then it reads connection from connection pool.

⇒ The JNDI properties to configure in configuration xml are

1) jndi.class → implementation class of JNDI api.
2) jndi.url → url to connect with registry
3) connection.datasource → It is key of Datasource object reference.

For example

hibernate.cfg.xml

```
<hibernate-configuration>
  <session-factory>
    <property name="jndi.class"> xxx </property>
    <property name="jndi.url"> xxx </property>
    <property name="connection.datasource"> xxx </property>
    ---
    ---
    ---
  </session-factory>
</hibernate-configuration>
```

Configuring a connection Pool in weblogic server 12.

⇒ When we download weblogic 12, a jar file with name wls-1213.jar is downloaded.

⇒ When we extract the jar file a folder is created with name wls-1213. Open Disk1 folder under it and then double click on install, to install the weblogic server.

⇒ After installation is finished, we will get c:\Oracle\Middleware\Oracle_Home\wlserver.

⇒ To work with weblogic server, first we need to create a domain like the following.

a) start → programs → Oracle → Oracle Home → Weblogic Server 12c → Tools → Configuration Wizard.

b) ⊙ Create a New domain (default) select

Domain Location: _ _ _ _ _ \test_domain

→ Next → Next Enter Name & password of administrator.

Name [hibernate]
Password [hibernate4]
Confirm Password hibernate4 → Next → Next → Next → Create → done.

c) The created domain is visible at below location.

c:|oracle|Middleware|Oracle-Home|user_projects|domains|test_domain.

→ start weblogic server.

2) open console page of weblogic server and enter username & password.

http://localhost:7001/console

Username : hibernate

Password : hibernate4

[Login] ↵

3) Left window panel → Domain structure → Services → expand → Data Sources →

New button → Generic Data Source →

Name [Test Data Source]

JNDI Name [Oraclejndi]

Database Type [oracle]

[Next]

Database Driver [Oracle's Driver (Thin) for Service Connections; Versions

[Next] → [Next]

4) Enter following connection properties,

Database Name [XE]

Host Name : [localhost]

Port : [1521]

DB User Name [System]

Password [tiger]

Confirm Password [tiger]

[Next]

5) click on Test Configuration button → [Next] → ☑ AdminServer → [Finish]

6) click on Test Data Source → ConnectionPool → change capacity values if required.(This step is optional).

*——*ConnectionPool*——*completes here*——*

⇒ In order to test whether hibernate is updating the connection from weblogic server connection pool or not, In one of the existing projects, open configuration file, remove connection properties and in place of them add jndi properties like below,

```
<!--jndi properties-->

<property name="jndi.class">weblogic.jndi.WLInitialContextFactory</property>
<property name="jndi.url">t3://localhost:7001</property>
<property name="connection.datasource">oraclejndi</property>
```

⇒ We need to add weblogic.jar to the build path of the project. (Weblogic.jar available in path: Oracle → Middleware → user_projects → wlserver → server → lib → weblogic.jar)

1/9/15

Hibernate Locking

⇒ In hibernate, two transactions (parallel/concurrent) can work on same data.

⇒ If a transaction updated the data and another parallel transaction is overriding the changes made by first transaction then conflict occurs. To resolve this conflicts hibernate has given locking mechanism.

⇒ If two parallel/concurrent transactions are working on two different data of same table(or) if transactions are sequentially executing then there is no chance of conflict and there is no need of applying hibernate locking.

⇒ To resolve the conflicts, hibernate has given 2 types of locking.

1) Optimistic locking
2) Pessimistic locking

2/9/15

optimistic locking

→ In this locking strategy, before committing a transaction, a transaction verifies whether version of an object in database and version of an object in Tx both are same or not.

⇒ If check revealed that there is a conflict then current transaction is going to be rollback.

⇒ In order to add version for an object, we need to apply versioning feature of hibernate.

⇒ To add version feature for an object, we need to do the following 2 changes.

1) In POJO class, we need to add version property. It is a variable of type int.
2) In hbm file, we need to add <version> tag immediately after closing <id> tag.

POJO class

Ex: public class student {

```
    private int studentId;
    private String studentName;
    private int marks;
*   private int version;
    }
```

student.hbm.xml

```
<hibernate-mapping>
    <class name="student" table="student">
      <id name="studentId" column="sid" type="int" precision="5" scale="0"/>
      <version name="version" column="version" type="int"/>
      ---
      ---
    </class>
</hibernate-mapping>
```

⇒ When a new object of student class is saved to database then hibernate inserts its version value as "0". when ever an object is updated then hibernate automatically inaments its version value by "1".

⇒ In optimistic locking if version of an object in a transaction is lower version than database object version then it is called a "stale object".

⇒ If a transaction is commiting lower version object changes then StaleObjectStateException will be thrown. StaleObjectStateException will be thrown.

Work Space : D:|user1

App1

- src
  - default package
    - Student.java
    - Client1.java
  - student.hbm.xml
  - hibernate.cfg.xml

Work space : D:|user2

App2

- src
  - default package
    - student.java
    - Client2.java
  - student.hbm.xml
  - hibernate.cfg.xml

Both projects are containing same code but names are different.

Client1.java

```
public class Client1{
    public static void main(String[] args)
    {
        SessionFactory factory = new Configuration().configure("hibernate.cfg.xml").
                                                    buildSessionFactory();
        Session session = factory.openSession();
        Transaction tx = session.beginTransaction();
        Student s = (Student) session.get(Student.class, 1);
        String str = JOptionPane.showInputDialog("Enter Name(User1)");
        s.setStudentName(str);
        tx.commit();
        session.close();
        factory.close();
    }
}
```

student.java

```
public class student{
    private int studentId;
    private String studentName;
    private int marks;
    private int version;

public int getStudentId(){
    return studentId;
}
public String getStudentName(){
    return studentName;
}
public int getMarks(){
    return marks;
}
public int getVersion(){
    return version;
}

public void setStudentId(int studentId){
    this.studentId = studentId;
}
public void setStudentName(String studentName){
    this.studentName = studentName;
}
public void setMarks(int marks){
    this.marks = marks;
}
public void setVersion(int version){
    this.version = version;
}
```

~~student.hbm.xml~~ student.java

```
public class Student{
    private int studentId;
    private String studentName;
    private int marks;
    private int version;
public int getStudentId(){
    return studentId;
}
public void setStudentId(int studentId){
    this.studentId = studentId;
}
public String getStudentName(){
    return studentName;
}
public void setStudentName(String studentName){
    this.studentName = studentName;
}
public int getMarks(){
    return marks;
}
public void setMarks(int marks){
    this.marks = marks;
}
public int getVersion(){
    return marks;
}
public void setVersion(int version){
    this.version = version;
}
```

~~hibernate.cfg.xml~~ client2.java

```
public class Client2{
    public static void main(String[] args){
SessionFactory factory = new Configuration().configure("hibernate.cfg.xml").
                                                buildSessionFactory();
    Session session = factory.openSession();
Student s = (Student) session.get(Student.class, 1);
Transaction tx = session.beginTransaction();
String str = JOptionPane.showInputDialog("Enter Name (user2)");
        s.setStudentName(str);
        tx.commit();
    session.close();
    factory.close();
    }
}
```

Execution steps

⇒ create a table with name student in Oracle.

```
sql> create table student (sid number(5) primary key,
                           sname varchar2(10),
                           marks number(5),
                           version number(3));

sql> insert into student values (1, 'xyz', 500, 0);

sql> commit;
```

⇒ Execute client1.java, it begin a transaction, reads student from table and its version is "0", then a dialog box will be display.

⇒ Execute client2.java, a transaction started, it reads student version having "0", then a dialog box displayed.

⇒ Now two transactions are parallely waiting.

⇒ Go to client1, enter the name in dialog box of user1 then click on [OK] button. This transaction has version "0" and in database also version is "0", so this transaction is committed and hibernate increments version of column by "1" in database.

⇒ Now go to client2.java, enter the name in dialog box of user2 then click on [OK], This transaction has version "0" and in database the version is "1", so this transaction is cancelled then StaleObjectStateException is thrown.

3/9/15

Timestamp feature for optimistic Lock

⇒ similar to versioning we can also use time stamp for optimistic lock. In case of

⇒ Before committing a transaction, it verifies time stamp values of an object and timestap value of database, both are same or not.

⇒ If same, then a transaction is committed it changes & in database timestamp is modified

⇒ If not same, then a transaction having stale object is roll back and stale object-state Exception is thrown.

*. hbm. xml

```
<hibernate-mapping>
  <class name="student" table="student">
    <id name="studentId" type="int" column name="sid" precision="5" scale="0"/>
    </id>
    <timestamp name="tstamp" column="tstamp" type="timestamp" />
    <property name="studentName" column="sname" length="10" />
    <property name="marks" column="marks" />
  </class>
</hibernate-mapping>
```

⇒ To add Timestamp feature in optimistic lock application then we need to the following two changes.

(1) In POJO class, we need to take a reference variable of type Timestamp.

(2) In hbm file, we need to configure <timestamp> tag immediately after closing </id> tag.

POJO:

```
public class student {
    private int studentId;
    private string studentName;
    private int marks;
    private Timestamp tstamp;
    // setters and getter methods
}
```

Pessimistic Locking

⇒ In this locking strategy, a transaction will get a lock on a record, while selecting that record, for updation.

⇒ If a transaction acquired a lock on a row for updating it then another parallel transaction can read that row, but it can not update the row, until lock is released by the first transaction.

⇒ In pessimistic lock, a row in table is not modified by any other transaction until a transaction who acquired a lock is completed. There is no question of stale object in the transaction.

⇒ A problem with pessimistic locking is a transaction should wait for a long time until a first transaction releases the lock.

⇒ In a pessimistic lock, we donot need either version or timestamp feature of hibernate

POJO class

Student.java

```
public class student{

    private int studentId;
    private string studentName;
    private int marks;
    private Timestamp tstamp;

  public int getStudentId()
   {
     return studentId;
   }

  public string getStudentName()
    { return studentName;
    }

 public int getMarks()
    { return marks;
    }

public Timestamp getTStamp()
    {
    return tstamp;
    }
}
```

```
public void setStudentId(int studentId){
        this.studentId = studentId;
    }
public void setStudentName(String studentName){
        this.studentName = studentName;
    }
public void setMarks(int marks){
        this.marks = marks;
    }
public void setTStamp(Timestamp tStamp){
        this.tStamp = tStamp;
    }
```

client1.java

```
public class client1{
    public static void main(string[] args){

SessionFactory factory = new Configuration().configure("hibernate.cfg.xml").
                                                buildSessionFactory();
Session session = factory.openSession();
student s = (student) session.get (student.class, 1, LockMode.UPGRADE_NOWAIT);
Transaction tx = session.beginTransaction();
String str = JOptionPane.showInputDialog("Enter Name (user1)");
        s.setStudentName(str);
        Thread.sleep(10000);
      S.O.P("object saved successfully");
       session.saveOrUpdate(s);
       tx.commit();
     S.O.P("record modified @ client1");
      session.close();
      factory.close();
  } }
```

student.hbm.xml

```
<hibernate-mapping>
<class name = "student" table = "student">
<id name = "studentId" type = "int" column name = "sid" precision = "5" scale = "0" />
</id>
<timestamp name = "tstamp" column = "tstamp" type = "timestamp" />
<property name = "studentName" column = "sname" length = "10" type = "string" />
<property name = "marks" column = "marks" />
</class>
</hibernate-mapping>
```

hibernate.cfg.xml

```
<hibernate-configuration>
<session-factory>
- - - -
- - - - -
<mapping resource = "com/s/h/config/student.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

Student.java

```
public class Student{

    private int studentId;
    private string studentName;
    private int marks;
    private Timestamp tstamp;

   //setters and getters
   - - -
   - - -

}
```

client2.java

```
public class client2{
 public static void main (String[] args){

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


    Session s = factory.openSession();
  Student s = (Student)session.get (Student.class, 1, LockMode. UPGRADE_NOWAIT);
  Transaction tx = session. beginTransaction();
  String str = JOptionPane.showInputDialog("Enter Name(user2)");
    s.setStudentName(str);
    session.saveOrUpdate(p);
    S.O.P("Object saved successfully");
     tx.commit();
    session. close();
   factory. close();
}}
```

student.hbm.xml

```
<hibernate-mapping>
<class name="student" table="student">
<id name="studentId" type="int" column name="sid" precision="5" scale="0"/>
</id>
<timestamp name="tstamp" column="tstamp" type="timestamp" />
<property name="studentName" column="sname" type="string" length="10"/>
<property name="marks" column="marks"/>
</class>
</hibernate-mapping>
```

hibernate.cfg.xml

```
<hibernate-configuration>
<session-factory>
---
---
<mapping resource="com/sih/config/student.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

How to call a procedure/function using hibernate

⇒ In hibernate, we don't have any seperate approach for calling procedure/function.

⇒ In hibernate also, we use CallableStatement of JDBC for calling a procedure/functi

⇒ To define CallableStatement logic of JDBC, we need to implement a class from Work Interface and we need to override execute() method and we need to define the CallableStatement logic in execute() method.

For ex:

```
public class MyWork implements Work
{
    @Override
  public void execute (Connection con) throws SQLException
   {
       //callablestatement logic
   }
}
```

⇒ In order to call execute() method of mywork class, we need to pass MyWork class object to doWork() method of session.

```
session.doWork (new MyWork());
```

SQL> ed functionname (already created in oracle)
↳ contains function code

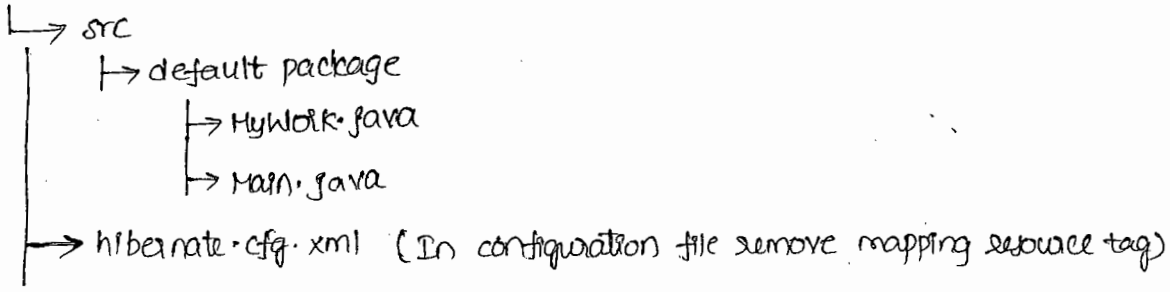
4/9/15

⇒ The following function in oracle reads input as employee Number and it returns bonus as output based on salary of employee.

```
⇔ create or replace function fun_bonus (eno number)
  return number is temp number(5);
                   bonus number(5);
  begin
    select sal into temp from emp where empno = eno;
      if temp <= 2000 then
         bonus := temp * 0.10;
    else if temp > 2000 and temp <= 6000 then
         bonus := temp * 0.20;
    else                                    return bonus;
  end if; bonus := temp * 0.30;
                                          end;
```

```
SQL > @ a2          (compile file a2)
  23 /              (total no.of lines in file)

  Function created.

SQL> ed a2          (edit the file already created)
        ↳ file contains code
```

Procedure Test



MyWork.java

```
public class MyWork implements myWork
{
    @override
    public void execute(Connection con) throws SQLException
    {
        CallableStatement cstmt = con.prepareCall("{? = call fun_bonus(?)}");
            cstmt.registerOutParameter(1, Types.INTEGER);
            cstmt.setInt(2, 7456);
            cstmt.execute();
            int bonus = cstmt.getInt(1);
            S.O.P("Bonus :="+bonus);
        cstmtsession.close();
    }
}
public class Main
{ public static void main(String[] args)
  { SessionFactory factory =new Configuration().configure("hibernate.cfg.xml").
        Session session = factory.openSession();    buildSessionFactory();
          session.doWork(new MyWork());
          session.close();
} }
```

Inserting an image using hibernate

⇒ In hibernate, to insert an image, in a POJO class we need to create a reference variable of Blob type.

⇒ An image is a collection of bytes, we need to send the bytes we need to convert bytes into Blob object.

⇒ When we insert an image then in database directly image is not saved. Instead its bytes are saved.

⇒ In the following example, we are going to insert an Employee class object with ID, Name & photo.

ImageTest
- src
  - default package
    - Employee.java
    - Insert.java
- hibernate.cfg.xml
- employee.hbm.xml

```
public class Employee
{
private Int employeeId;
private string employeeName;
Private Blob photo;
   //setter & getter methods
----
----
}
```

employee.hbm.xml

```
<hibernate-mapping>
   <class name="Employee" table="employees">
    <id name="employeeId" column="empid"/>
    <property name="employeeName" column="empname" length="10"/>
    <property name="photo" column="photo" type="blob"/>
   </class>
</hibernate-mapping>
```

hibernate.cfg.xml

```
<hibernate-mapping>
<session-factory>
---
---
<mapping resource="com/sl/h/config/employee.hbm.xml"/>
</session-factory>
</hibernate-mapping>
```

Insert.java

```
public class Insert
{ public static void main(String[] args) throws Exception {
    SessionFactory
       Session
    //create File Object
    File file = new File("D:/pichai.jpg");
    find no. of bytes in file
    int length = (int) file.length();
    // create byte[]
    byte bytes[] = new byte[length];
```

```
        //Create FileInputStream object
      FileInputStream fis = new FileInputStream(file);
         //Read bytes from file
           fis.read(bytes);
       //Create Blob object
      Blob blob = Hibernate.createBlob(bytes);
        //Create Employee object
       Employee e = new Employee();
         //set values
         e.setEmployeeId(11011);
         e.setEmployeeName("Sunder");
         e.setPhoto(blob);
      //save
    Transaction tx = session.beginTransaction();
       session.save(e);
         tx.commit();
        session.close();
       factory.close();
   }
}
```

select.java

```
public class RetrevClient {
    public static void main(String[] args) throws Exception{
         Session ses = HibernateUtil.getSession();
        //EmpDetails eb = (EmpDetails) ses.load( Employee.class, new Integer(1001));
           String name = eb.getEmployeeName();
          Blob photo = eb.getPhoto();
        HibernateUtil.closeSession();
       }
  }
```

5/9/15

enum type

⇒ An enum is a special datatype, whose variable can hold one of the fixed set of constant values defined for that type.

⇒ "enum" is a keyword to create an enum type.

⇒ In an enum type, by default all the fields (variables) are public, static & final. so the fields are written in uppercase letters.

⇒ Apart from constant fields, we can also define a private constructor and public methods in an enum type.

⇒ For example, in a project we want to create a variable and its value must be one of whether it NORTH/SOUTH/EAST/WEST then we need to create an enum type in the project like the below,

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST
}

Direction d = Direction.EAST; //assign a value to 'd' of type Direction.
```

⇒ When an enum type is compiled, it is automatically extended from java.lang.Enum class, but not from java.lang.Object.

⇒ While creating an enum type, we can assign values to the constant fields, with in a paranthesis.

For example:

```
public enum Signal
{
    RED (10), YELLOW (20), GREEN (30); //values are Int type.
}

public enum Day
{
    SUNDAY ("SUN"), MONDAY ("MON"), TUESDAY ("TUE"), WEDNESDAY ("WED")
}
```

⇒ For the constants of enum type we can assign either integer, string, float etc..

Annotations Introduction

⇒ In java when we are defining the source code to make it was easily understand to other java developers. We define all possible comments either in documentation style or in ordinary style.

⇒ With the help of comments define in source code other developers can understand the code easily. so we call this comment as a metadata.

⇒ While creating java elements like classes or variables or methods or constructors etc., we add modifiers, in order to tell meta data of that element to a java compiler.

For example :
```
public final class A
{
 --
 --
}
```

* In the above, final modifier tell a meta data about class A to the compiler that it can not be extended.

⇒ While working with technologies, tools & frameworks, to tell the metadata of classes to the containers, we prepare seperate xml files.

⇒ As the no.of technologies and frameworks are increasing, the no.of xml files and sizes of xml files are also got increased.

⇒ Due to heavy use of xml, the burden on java developers is also got increased.

⇒ As a solution for the problem of xml's, we got a annotations in java from jdk5.

⇒ Finally we have 4 ways of defining meta data in java,

(1) comments
(2) modifiers
(3) xml
(4) annotations.

⇒ The similarity in comments, modifiers and annotations is, the meta data is defined in source code only and in case of xml, a seperate file is needed to be constructed.

#91s

How annotations are created in Java?

A)⇒ Sun Microsystems provided a standard specification called JavaConfigAnnotations, for creating annotations in JAVA.

⇒ JavaConfigAnnotations are given in java.lang.annotations package.

⇒ Annotations	enum types
1) @) Target	1) ElementType
2) @) Retention	2) RetentionPolicy
3) @) Documented	

1) @) Target

⇒ While creating annotation, it must be specified that, for which type of element an annotation is applicable.

⇒ In order to specify element type, @) Target annotation is used.

⇒ for @) Target annotation, we need to pass ElementType as parameter.

⇒ ElementType is an enum and it has the following constant fields.

① FIELD → It indicates that annotation is applicable for variable

② METHOD → applicable for method

③ CONSTRUCTOR → applicable for constructor

④ PARAMETER → applicable for method parameter

⑤ TYPE → applicable for class / interface

⑥ ANNOTATION_TYPE → applicable for annotation

Ex:1)

```
@Target(ElementType.METHOD)
public @Interface Sathya
{
  __
  __
}
```

(@Interface: Interface is the keyword; Sathya: uppercase)

→ @) Satya annotation is application for methods.

Ex(2) @Target ({ ElementType.FIELD, ElementType.METHOD, ElementType.CONSTRUCTOR })

```
public @Interface Sathya
{ --
}
```

→ @Sathya annotation is application for variables, methods & constructors.

2) @Retention

⇒ This annotation is used to specify, at what point of time an annotation should be discarded from the code.

⇒ To this @Annotation Retention we need to pass a RetentionPolicy as a parameter.

⇒ RetentionPolicy is an enum & it has following 3 constant fields.

1) SOURCE → It indicates that, annotation should be discarded during compilation time by the compiler. The annotation is not included in .class file.
2) CLASS → It indicates that annotation is need to be included in class file and it is discarded while loading a class into JVM by a class loader.
3) RUNTIME → It indicates that annotation should be included in .class file and it should be visible in JVM also.

Note If RetentionPolicy is not set for an annotation while creating it then by default RetentionPolicy is CLASS.

⇒ For example

```
@Target (ElementType.METHOD)
@Retention(RetentionPolicy.CLASS)
public @Interface Sathya
{
  --
}
```

⇒ @Sathya annotation is application for methods and it is discarded at the time of loading a class file into JVM.

3) @Documented

⇒ This annotation is to specify wheather an annotation is visible for javadoc tool for generating documentation or not.

What an annotation contains ?

⇒ An annotation contains elements to pass meta data.

⇒ An element in an annotation looks like a method, but it is not a method and it looks like a variable, but it is a variable. It is an element.

⇒ An element can have a default value.

⇒ only the following data types are allowed for elements.

1) Primitive Data types
2) String
3) Class
4) enum type

For ex:

```
@Target( ElementType.METHOD)
@Retention ( RetentionPolicy. RUNTIME)
public @interface Sathya
{
   String courseName();
   int fees() default 0;
}
```

@Sathya → invalid

@Sathya (courseName = "JAVA") → valid

@Sathya (fees = 1000) → invalid

@Sathya (courseName = "JAVA", fees = 1000) → valid

8/9/15

Hibernate Annotations (Hibernate JPA) [Java Persistance API]

⇒ As part of EJB 3 technology, sun micro system is releases a specification with the name of JPA which contains annotations for mapping a POJO class with an a database table.

⇒ To allow the developers easily to migrate from ORM TOOL to another, ORM tools vendors. also uses annotations of JPA for mapping POJO class to db table.

⇒ If every ORM Tool provides different annotations for mapping then burden on a programmer is increased while migrating from one ORM tool to another.

⇒ In order to reduce the developers burden all ORM tools vendors are agreed to follow JPA annotations for mapping.

⇒ JPA hay a single package javax.persistence. because of Hibernate uses annotations

⇒ of JPA for mapping Hibernate annotations is called Hibernate JPA.

⇒ A difference between developing Hibernate appln with out annotations with annotations is, in with out annotations, we use hbm.xml file for mapping and with annotations we remove hbm.xml file.

⇒ Annotations of java are having a drawback, the drawback is annotations are added in source code and in feature if any modifications are required in annotation then we need to recompile source code, reload the project, restart the sewer.

Basic annotations used for mapping.

@Entity
@Table
@Id
@Column
@Transient

⇒ To save an object a class is save in db then on top of class @Entity is mandatoly. By adding this annotation a class is made as an Entity bean (POJO class).

⇒ @Table annotation is to map a class with table.

⇒ @Id is indicates a property is a primary key

⇒ @Column is to map a variable with a column

⇒ @Transient : Make the fields as non-persistence fields & etc..

⇒ @Entity ~~are~~, @Table are class level annotations

⇒ @Id and @Column can be applied on top of variable are on top of method.

for Ex

Annotations contains elements

```
@Entity
@Table (name="student")
public class Student
{
    @Id
    @Column (name="sid")
    private int studentId;
    @Column( name= "sname", length="10")
    private string studentName;
    @column( name="marks")
    private int marks;
public void setStudentId ( int studentId){
        this.studentId = studentId; }
public void setStudentName( string studentName){
        this.studentName = studentName;
        }
public void setMarks (int marks){
        this.marks =marks;
        }
}
```

Note If class name and table name are same then use @Table annotation.

9/9/15

Annotation Example 1

Annotation Example
- src
  - default package
    - Book.java
    - Main.java
- hibernate.cfg.xml

Book.java

```
import javax.persistence.*;
@Entity
@Table(name="Book")
public class Book {
    @Id
    @column
    private int bookId;
    @column
    private String bookName;
    @column
    private double price;

    //setters and getters
}
```

hibernate.cfg.xml

```
<mapping class="Book" />
```

Main.java

```
public class Main
{
    public static void main(
    {
        SessionFactory factory = new Configuration().configure("hibernate.cfg.xml").
            Session session = factory.openSession(); buildSessionFactory();
            Book book = new Book();
                book.setBookId(10101);
                book.setBookName("JAVA");
                book.setPrice(600);
```

```
Transaction tx=session.beginTransaction();
            session.save(book);
              tx.commit();
            session.close();
            factory.close();

        }
    }
```

Annotations for Inheritance Mapping

⇒ Hibernate has provided @Inheritance for selecting Inheritance strategy.

1) @Inheritance(strategy = InheritanceType.SINGLE_TABLE) [Table per class] (InheritanceType: enum)
2) @Inheritance(strategy = InheritanceType.TABLE-PER-CLASS) [Table per concrete class]
3) @Inheritance(strategy = InheritanceType.JOINED) [Table per subclass]

@Inheritance is a class level annotation.

⇒ In case of Table per class inheritance strategy, database table must contain 1 extra column called "Discriminator column".

⇒ In a Discriminator column, a discriminator value is stored, which indicates the subclass whose object is inserted.

⇒ In annotations, to tell the hibernate about discriminator column name, annotation is @DiscriminatorColumn.

⇒ To set discriminator value for a subclass, annotation is @DiscriminatorValue.

⇒ To execute Table per class strategy with annotations per payment application then we need to modify payment, credit card & cheque classes with annotation like the following Payment.java

```
@Entity
@Table(name="payment")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="pmode", discriminatorType=DiscriminatorType.STRING, length=6)
public abstract class Payment
{
  @Id
  @Column(name="payid")
  private int paymentId;
```

```
@Column (name ="amount")
private double amount;
@Temporal ( TemporalType.DATE)
@column (name ="pdate")
private Date paymentDate;
// setters and getters
}
```

Credit Card.java

```
@Entity
@DiscriminatorValue ( value ="CR")
public class CreditCard extends Payment
{
    @Column (name ="ccno")
    private int cardNumber;
    @column ( name ="cctype", length=8)
    private string cardType;
    //setters and getters
}
```

cheque.java

```
@Entity
@DiscriminatorValue (value ="CH")
public class cheque extends Payment
{
    @ column (name ="chno");
    private int chequeNumber;
    @column (name ="chtype", length=8)
    private string chequeType;
} // setters and getters.
```

hibernate.cfg.xml

```
<mapping class="com.s.h.model.Payment"/>
<mapping class="com.s.h.model.CreditCard"/>
<mapping class="com.s.h.model.Cheque"/>
```

(2) Table per Concrete class strategy

⇒ If we want to convert payment Application as Table per concrete class strategy then we need to modify the classes like the following.

//Payment.java

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Payment
{
    @Id
    @Column(name="payid")
    private int paymentId;
    @Column(name="amount")
    private double amount;
    @Column(name="pdate")
    private Date paymentDate;
    //setters and getters
}
```

//CreditCard.java

```
@Entity
@Table(name="credit_table")
public class CreditCard extends Payment
{
    @Column(name="ccno")
    private int cardNumber;
    @Column(name="cctype", length=8)
    private String cardType;
    //setters and getters
}
```

//Cheque.java

```
@Entity
@Table(name="cheque_table")
public class Cheque extends Payment
{
    @Column(name="chno");
    private int chequeNumber;
    @Column(name="chtype", length=8)
    private String chequeType;
    //setters and getters;
}
```

10/9/18

Table per subclass

⇒ When we are applying or using table per subclass strategy with annotations then, in db there will be a table per parent class and seperate table per each child class.

⇒ In each child class table there will be a primary key and it is also used as foreign key for the relationship so in order to tell the hibernate that a column is primary key in the table and foreign key for the relationship, we use @PrimaryKeyJoinColumn annotation.

// payment.java

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

Public abstract class Payment
{
   @Id
   @column (name="payId")
   Private int paymentId;
   @column( name="amount")
   private double amount;
   @column (name="pdate")
   private Date paymentDate;

public int getPaymentId() {
    return paymentId;
   }
public double getAmount(){
    return amount;
   }
public Date getPaymentDate(){
    return paymentDate;
   }

public void setPaymentId(int paymentId){
    this.paymentId = paymentId;
   }
public void setAmount(double amount){
    this.amount = amount;
   }
public void setPaymentDate(Date paymentDate){
    this.paymentDate = paymentDate;
   }
}
```

// Credit Card.java

```
@Entity
@Table (name = "credit_table")
@PrimaryKeyJoinColumn(name = "pid")
public class CreditCard extends Payment
{
    @Column(name = "cno")
    private int cardNumber;
    @Column(name = "cctype", length = 8)
    private String cardType;
    //setters and getters

}
```

Cheque.java

```
@Entity
@Tablename(name = "cheque_table")
@PrimaryKeyJoinColumn(name = "pid")
public class Cheque extend Payment
{
    @Column(name = "chno")
    private int chequeNumber;
    @Column(name = "chtype", length = 8)
    private String chequeType;
    //setters and getters

}
```

Annotations for generator classes

⇒ To add a specific Generator to the Id, we use the following annotations we use one from JPA. one from Hibernate.

@GenericGenerator → Hibernate
@GeneratedValue → JPA

⇒ @GenericGenerator Annotation is to select a generator strategy to give an Alias name to the strategy.

⇒ @GeneratorValue Annotation is to add a 'GeneratorId'.

⇒ For example, we can add assigned Generator to the 'Id' like the following.

```
@Entity
@Table(name="student")
public class student
{
  @GenericGenerator(name="g1" strategy="assigned")
  @Id
  @GeneratedValue(generator="g1")
  @Column(name="sid")
  private int studentId;
  ---
}
```

⇒ For example sequence generator can be added 'Id' like the following,

```
public class student{
  @GenericGenerator(name="g1" strategy="sequence", parameters=@Parameter
                    (name="sequence" value="mysequence"))
  @Id
  @GeneratedValue(generator="g1")
  @Column(name="sid")
  private int studentId;
  -----
  ---
}
```

⇒ For example we can add hilo Generator for 'id' like the following

```
public class student{
    @GenericGenerator (name = "g1", strategy = "hilo",
                parameters = { @parameter (name = "table", value = "mytable",
                               @parameter (name = "column", value = "column1",
                               @parameter (name = "max-lo", value = "10" } )

    @Id
    @GeneratedValue(generator = "g1")
    @Column (name = "sid")
    private int studentId ;
    - - - -
    - - - -
    - - - - -
}
```

Annotations based Association Mapping | Relationship

@OneToMany → To Build association with the suppot of collection property

@JoinColumn → To specify FK column

Enums

FetchType.Lazy | EAGER → For enabling Lazy | EAGER loading

CascadeType. ALL | PERSIST | MERGE | REFRESH | REMOVE | DETACH

(To Specify the cascading Types)

MERGE → cascade the merge operations done on main object to the associated object

DETACH → If main object becomes Detached object the associated object also becomes Detached object

REFRESH → If main object is reloaded from DB, the associated child object will also be reloaded from DB.

Example

User_Table (parent)

↦ user_id (pk)

↦ first_name

Phone_numbers (child)

↦ phone (pk)

↦ number_type

↦ uniId (fk)

User.java

```
package com.s.h.domain;
    @Entity
    @Table (name = "user_table")
    public class User
    {
      @user @Id
      @Column (name = "user_id")
      public int userId;
      private string firstName;
      @OneToMany (targetEntity = PhoneNumber.class, cascade = cascadeType.ALL,
                  fetch = FetchType.LAZY, orphanRemoval = true)
      @JoinColumn (name = "unId", referencedColumnName = "user_id")
       private Set<PhoneNumber> phones;
        //setters and getters
       ---
       ---
      }
```

PhoneNumber.java

```
package com.s.h.domain;
    @Table (name = "phone_numbers")
    @Entity
    public class PhoneNumber
    {
       @Id
       private long phone;
       @Column (name = "number_type")
       private string numberType;
      //setters and getters
      ---
      ---
      }
```

⇒ While working with Annotations the default FetchType in OneToMany is FetchType.LAZY & default fetchType in ManyToOne is FetchType.EAGER.

HBProj4(Anno-OneToMany-Uni)

```
|→src
   |→com.s.h.cfgs
        |→hibernate.cfg.xml
   |→com.s.h.domain
        |→User.java, PhoneNumber.java
   |→com.s.h.dao
        |→OneToManyDao.java, OneToManyDaoFactory.java, OneToManyDaoImpl.java
   |→com.s.h.utility
        |→HibernateUtil.java
   |→com.s.h.test
        |→Main.java
```

Working with java.util.List type Collection

⇒ In OneToMany association if the collection type is java.util.List we need to maintain an extra index column in child table and we can use
@IndexColumn (Hibernate Annotation that is deprecated) or
@OrderColumn (JPA Annotation) to specify that index column

Example

User.java

```
@OneToMany (targetEntity = PhoneNumber.class, cascade = CascadeType.ALL,
                    orphanRemoval = true)
@JoinColumn (name = "unId", referencedColumnName = "user_id")
@IndexColumn (name = "lst_indx") → Deprecated or
@OrderColumn (name = "lst_indx")
private List<PhoneNumber> phones;
```

DB Tables : User_table (parent)

User_id (pk)	first_name
101	rafa

Phone_numbers (child)

Phone	number_Type	unId	lst_index (Index column)
9999999999	office	101	0
8888888888	residence	101	1

NOTE: As of now there is no Annotation alternate for <bag> tag.

Taking collection type as java.util.Map

⇒ In One To Many Association if the collection type is java.util.Map then the child table must contain an extra index column to hold the keys of map elements. We need to cfg this column by using @MapkeyColumn .annotation (JPA).

Example

User.java

```
@OneToMany(---)
@JoinColumn(---)
@MapkeyColumn (name="map_index")
private Map<String, PhoneNumber> phones;
```

DB Tables

User_Table

User_id	first_name
101	raja

Phone_numbers

Phone	number_type	uniId(fk)	map_indx
---			ph1
---			ph2

Annotations based ManyToOne uni-directional (child parent)

Annotations

@ManyToOne : To refer parent class object being from child class
@JoinedColumn : To specify FK column

Enums

CascadeType, FetchType, --

Example

Department.java(parent)

```
@Table (name="Department")
@Entity
public class Department
{   @Id
    private int deptno;
    private String deptname,depthead;
 //setters and getters
}---
```

EmpDetails.java

```
@Table (name = "EmpDetails")
@Entity
public class EmpDetails
{
  @Id
  private int eno;
  private string ename;
  private long salary;
  @OneToMany( targetEntity = Department.class , cascade = CascadeType.ALL, fetch = FetchType.
                                                                                    EAGER)
  @JoinedColumn (name="deptno", referencedColumnName ="deptno")
  private Department dept ;
  //setters and getters
  ---
}
```

DB Tables

Department(parent)

deptno(pk)	deptname	depthead
1001	accounts	smith

EmpDetails (child)

eno	ename	salary	deptno (fk)
101	raja	9000	1001
102	ravi	9000	1001

OneToMany Bi-Directional

Annotations

@OneToMany : To make parent class property to hold set of child objects

@ManyToOne : To make child class property to hold one parent object
(useful to make multiple child objects to hold one parent class)

@JoinColumn : To specify fk column

mappedBy param

⇒ In Bi-directional association we need to specify FK column at parent class and also at child class by using special annotation like @JoinColumn. If we want to specify that special annotation only at 1 side to reflect that to other side we need to use "mappedBy" param specifying the other side special property.

⇒ This parameter is available only in @OneToOne, @OneToMany, @ManyToMany annotation

Example

User.java (parent)

```
@OneToMany(targetEntity = PhoneNumber.class, cascade = CascadeType.ALL,
           fetch = FetchType.LAZY, mappedBy = "user")
private Set<PhoneNumber> phones;
```

PhoneNumber.java (child)

```
@ManyToOne(targetEntity = User.class, cascade = CascadeType.ALL, fetch = FetchType.LAZY)
@JoinColumn(name = "unId", referencedColumnName = "user_id")
private User user;
```

DB Tables

User_table (parent)
- user_id (PK)
- first_name (varchar2)

Phone_numbers (child)
- phone (PK)
- number_type
- unId (fk)

Annotations based Many To Many Associations

Annotations

@ManyToMany : To build Many To Many Association

@JoinColumn : To specify owning side FK column and reverse side FK column

@JoinTable : To specify 3rd table cfgs

Programmers (table1)

↦ pid (pk)

↦ pname

↦ salary

Projects (table2)

↦ projid (pk)

↦ projname

Programmers_projects (table3)

↦ programmer_id (fk)

↦ project_id (fk)

Programmer.java (parent class)

```
@Entity
@Table(name="programmers")
public class Programmer
{
   @Id
   private int id;
   private String pname;
   private long salary;
@ManyToMany(targetEntity = Project.class, cascade = CascadeType.ALL, fetch=FetchType.LAZY)
@JoinTable(name="programmers_projects", joinColumns = @JoinColumn(name="
programmer_id", referencedColumnName="pid"),
inverseJoinColumns = @JoinColumn(name="project_id", referencedColumnName="projid"))
//setters and getters
---
---
}
```

Project.java

```
@Entity
@Table(name="projects")
public class Project
{
   @Id
   private int projId;
   private String progname;
  @ManyToMany(targetEntity = Programmer.class, cascade=CascadeType.ALL,
        mappedBy="projects")
  private set<Programmer> programmers = new HashSet<Programmer>();
  //setters and getters
  - --
  - --
}
```

Working with @CollectionId (Alternate to <idbag> tag)

⇒ While working with ManyToMany association the 3rd table / join table / cross table does not contain uniqueness because fk columns contains duplicate values to build the association. To get uniqueness in 3rd table we can add 1 special index col and we can cfg Id generator to fill the values in the index column (other than assigned)

⇒ To cfg this special index columns through xml mapping we can use <idbag> tag, similarly we can use Hibernate supplied annotation @CollectionId for the same.

Example

Programmer.java

```
@ManyToMany(--)
@JoinTable(---)
@GenericGenerator(name="gen1", strategy="increment")
@CollectionId(columns=@Column(name="proj-prgmr_index"), type=@Type(type="int"),
    generator="gen1")
private List<Project> projects =new ArrayList<Project>();
```

Programmers_projects (join table)

Programmer_id	project_id	proj-prgmr-index
101	1001	1
101	1002	2
102	1002	3

Annotations based One To One Association

⇒ It can be implemented in 2 ways

a) As One To One FK

b) As One To One PK

1) One To One FK

Annotations:

a) @ManyToOne with unique="true", not-null="true" (To build one to one fk association)

b) @JoinColumn (To specify FK column)

Example (One To one FK uni-directional)

Person (parent)

↦ person_id (PK)

↦ first Name

↦ last Name

↦ age

License (child)

↦ license_id (PK)

↦ type

↦ valid_from (date)

↦ valid_to (date)

↦ license_holder (fk)

Person.java

```
@Entity
@Table
public class Person
{
    @Id
    @Column (name = "person_id")
    private int id;
    private String firstName;
    private String lastName;
    private byte age;
  //setters and getters
  ---
  ---
}
```

License.java (child)

```
@Entity
@Table
public class License
{
  @Id
   @Column (name = "license_id")
   private int lid;
   private String type;
   @Column (name = "valid_from")
   private Date datefrom;
  @Column(name="valid_to")
  private Date dateto;
@ManyToOne (targetEntity = Person.class, unique = "true", not-null = "true")
@JoinColumn(name = "license_holder", referencedColumnName = "person_id")
private Person licenseholder;
//setters and getters
---
}
```

⇒ If we use @OneToOne with @JoinColumn then it becomes OneToOneFK association.
If we use @OneToOne with @PrimaryKeyJoinColumn then it becomes OneToOnePK association.

2) OneToOne PK

Annotation

@OneToOne : To build OneToOne Association

@PrimaryKeyJoinColumn : Uses the PK column value of parent table as PK column value of child table with out having any FK constraint support.

Example

OneToOne PK Bi-directional

⇒ Here FK column is not required but "foreign" generator cfg is required to copy the identity value of parent object as the identity value of child object.

DB Table

Student
- ⟼ sid(PK)
- ⟼ name
- ⟼ address

Lib_Membership
- ⟼ lid(PK)
- ⟼ doj(date)

Student.java

```
@Entity
@Table
public class Student
{
  @Id
  @Column(name="sid")
  @GenericGenerator(name="gen1", strategy="increment")
  @GeneratedValue(generator="gen1")
  private int id;
  private String name, address;
  @OneToOne(targetEntity=LibraryMembership.class, cascade=CascadeType.ALL,
                          mappedBy="studentDetails")
  private LibraryMembership libraryDetails;

  //setters and getters
  ---
}
```

LibraryMembership.java

```
@Entity
@Table (name="lib_membership")
public class LibraryMembership
{
    @Id
    @Column(name="lid")
    @GenericGenerator( name="gen1", strategy="foreign",
        parameters= {@Parameter(name="property", value="studentDetails")})
    @GeneratedValue( generator="gen1")
    private int id;
    @Column(name="doj")
    private Date joiningDate;
    @OneToOne (targetEntity=Student.class, cascade=CascadeType.ALL)
    @PrimaryKeyJoinColumn(name="lid", referencedColumnName="sid")
    Private Student studentDetails;
    // setters and getters
    ---
}
```

Q What is the difference b/w FetchType and FetchMode in annotations ?

A FetchType specifies weather object should be loaded lazily/eagerly whereas FetchMode specifies the no. of select SQL queries that should be generated while loading the objects. Use the HB supplied @fetch annotations to specify FetchMode & use fetch attribute of various association annotations to specify 'FetchType'.

Example

In user.java

```
@OneToMany(targetEntity=PhoneNumber.class, fetch=FetchType.LAZY/EAGER,
        cascade=CascadeType.ALL)  |---? //fetchType
@JoinColumn(---)
@Fetch(FetchMode.SELECT/SUBSELECT/JOIN) |---? //fetchMode
Private Set<PhoneNumber> phones;
```

NOTE : We must use Criteria API logic in order to take the benefit of FetchMode.

Second Level Caching Hibernate | L2 | Global | Level2 | sessionFactory cache

⇒ Hibernate supports two levels of caching to reduce nlw round trips between client & server.

(1) Level1 | L1 | First Level | Local Machine

(i) It is built in cache associated with session object

(ii) It is 1 per session object

(2). Level2 | L2 | Second Level | Global cache

(i) It is configure cache associated with sessionfactory object

(ii) It is 1 per sessionfactory object

⇒ When HB persistence logic loads the object

a) Searches for that in L1 cache if available takes that object, if not available

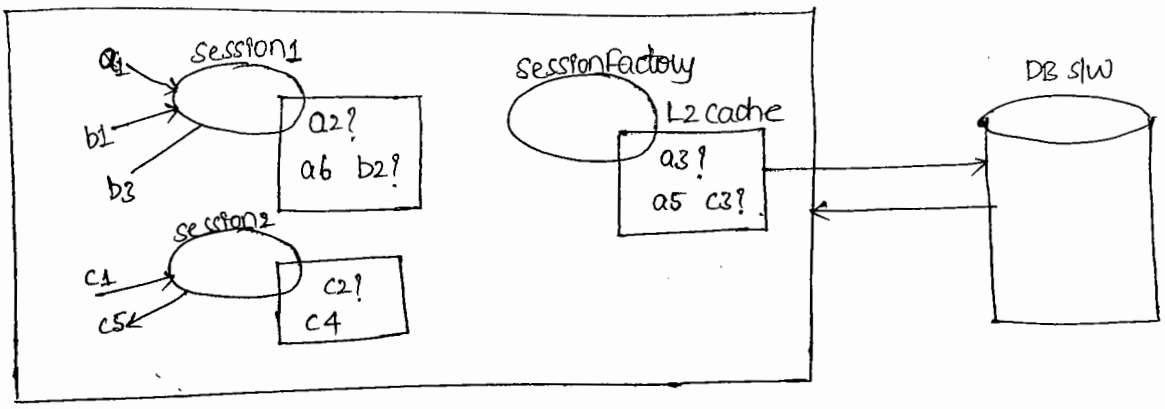
b) Searches for that in L2 cache if available takes that object, if not available

c) Gets the object by selecting record from DB table

d) Keeps that object in L2 cache

e) Keeps that object in L1 cache

f) Gives that object back to clientApp



NOTE

All the 3 requests are same requests.

⇒ Second Level cache is configurable caches. The following second level cache slwls are available. They are

a) Ehcache (Easy HB)
b) Swarm cache
c) Jboss Tree Cache
d) OS Cache
e) Tangosol coherence cache (commercial)

⇒ We need to specify Concurrency strategy towards placing objects & retrieving objects from L2 cache They are

a) Read-only
b) Read-write
c) Non strict read-write
d) Transactional

⇒ For more info various second level caches & concurrency strategy for ref page no: 190 to 192.

⇒ To cfg second level cache we must know the cache provider class name

EhCache → EhCacheProvider (Old) | EhCacheRegionFactory (new version) 4.x

OsCache → OsCacheProvider

and refer page no: 191

⇒ To control second level cache factory.close() → closes SessionFactory & also closes L2 cache

factory.evict(classname) : → Removes the object of class from L2 cache. |*|

factory.evictCollection() : → Removes collection object from L2 cache.

*| → Deprecated.

⇒ We can also use Cache.evict() and Cache.evictCollection() for the same.

⇒ We can get cache object by using factory.getCache() method.

Example to cfg SecondLevel (EhCache) in HibernateApp

1) Keep any HBApp ready (FirstApp)
2) Add the following additional jar files

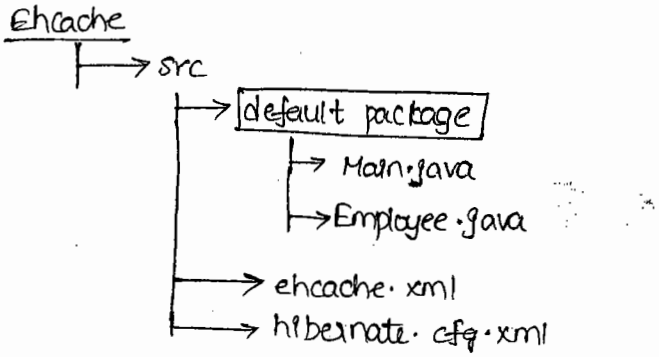
HB 4.3 home | lib | optional | EhCache all (jars) (3 jar files)

3) Add the following entries in HB cfg file to enable L2 cache

⇒ The most widely used level2 cache is EasyHibernate cache (EHcache).

⇒ In order to do settings for a level2 cache we need to create Ehcache.xml file.

⇒ Along with hibernate jars, we need to add a third party jar ehcache-1.2.3.jar



Employee.java

```
@Entity
@Table(name="emp")
@cache (usage = CacheConcurrencyStrategy.READ_ONLY
public class Employee
{
    @Id
    @Column(name="empno")
    private int employeedId;
    @Column(name="ename")
    private String employeeName;
    @Column(name="deptno")
    private int deptNumber;
    @column(name="sal")
    private int employee_Sal;
    //setters and getters methods
}
```

⇒ In hibernate.cfg.xml, we need to configure the following two cache properties.

```
<property name="cache.provider_class">org.hibernate.cache.EhCacheProvider</>
<property name="cache.use_second_level_cache">true</property>
```

Ehcahe.xml

```
<ehcache>
  <defaultCache maxElementsInMemory="100" eternal="false"
    timeToIdleSeconds="120" timeToLiveSeconds="200" />

  <cache name="Employee" maxElementsInMemory="10" eternal="false"
    timeToIdleSeconds="8" timeToLiveseconds="200" />

</ehcache>
```

⇒ In the above xml, cache settings are defined for objects of Employee and for other objects.

⇒ IdleSeconds indicates waiting time before an object is going to be removed from cache and liveseconds indicates the complete life time of an object, before it is going to be deleted from cache.

⇒ To make IdleSeconds & liveSeconds as working, we need to disable eternal by assigning eternal="false".

Main.java.

```
public class Main
{
    public static void main(String args[])
    {
        SessionFactory factory = new Configuration().configure().buildSessionFactory();

        Session ses1 = factory.openSession();
        Session ses2 = factory.openSession();
        Session ses3 = factory.openSession();

Employee e1 = (Employee) ses1.get(Employee.class, 7900);
        S.O.P (e1.getEmployeeName());
    ses1.clear();
 S.O.P(" session1 is cleared from cache ");
Employee e2 = (Employee) ses1.get(Employee.class, 7900);
    S.O.P (e1.getEmployeeName());
    S.O.P("======");
 try{
      Thread.sleep(6000);
    }
 catch(Exception e){
                    }

Employee e2 = (Employee) ses2.get(Employee.class, 7900);
        S.O.P (e2.getEmployeeName());
   S.O.P ("======");
  try{
      Thread.sleep(5000);
    }
  catch(Exception e){
                    }

Employee e3 = (Employee) ses3.get(Employee.class, 7900);
   S.O.P (e3.getEmployeeName());
```

```
        ses1.close();
        ses2.close();
        ses3.close();
        factory.close();
    }
}
```

o/p

Hibernate: select employee0_.empno as empno0_0_, employee

MILLER

session1 is cleared from cache.

MILLER

MILLER

MILLER

query cache

⇒ In query caching, a query its parameters and result of the query are stored in cache, when a query is executed for first time when same query with same parameters is executed for next time then result will be sent to the application from cache.

⇒ Query cache is not a seperate cache. It is a part of level2 cache.

⇒ To enable query caching, we need to add the following property in cfg.xml file.

<property name="cache.use_query_cache">true</property>

⇒ To store a query, its parameters and its results in cache, we need to set cachable as true, by calling setCacheable() method.

For ex:
```
Query query = session.createQuery("from Employee e where e.deptNumber=?");
query.setParameter(0,20);
query.setCacheable(true);
List list = query.list();
```

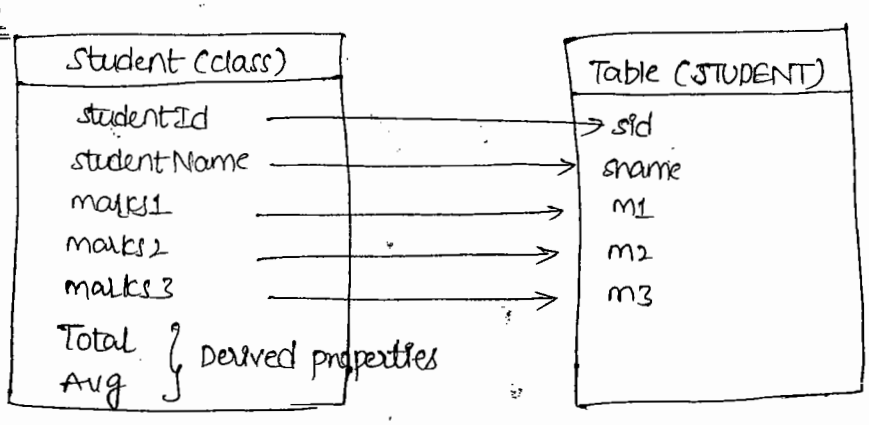
Q Is number of properties in POJO class and number of columns in table should be same?

A NO. Either POJO class can have more properties or a table can have more columns.

Q What is partial & full object mapping?

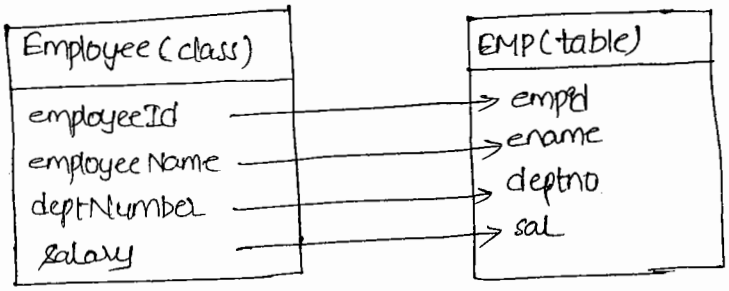
A If only some properties of a POJO class are mapped to columns of a table then it is partial object mapping and if all properties of POJO class are mapped to the columns of table then it is called full object mapping.

Ex1



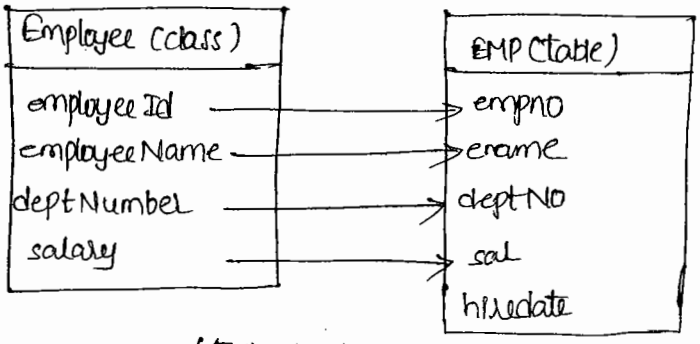
(partial object Mapping)

Ex2



(Fully object Mapping)

Ex3.



(Fully object mapping)

19/9/15

Hibernate filter

⇒ Hibernate filters is a concept added in Hibernate 3.6.

⇒ In an application, if we want to execute same query sometimes with condition and sometimes with out condition then we need to construct that query for multiple times.

⇒ Hibernate filters is a concept, which is used to seperate the condition from the query.

⇒ If we want to execute a query with condition then we enable the filter and if we want to execute the query with out condition then we disable the filter.

⇒ In Hibernate filters, we can configure a filter condition in 2 ways,

1) using hbm.xml file
2) using annotations

⇒ In hbm.xml, first we need a filter definition by using <filter-def> tag. This definition contains configuration of one or more filter parameters.

⇒ To add a filter at class level, we need <filter> tag under <class> tag.

For ex:

1) flight.hbm.xml

```
<hibernate-mapping>
  <class name="flight" table="flight">
    ---
    <filter name="myfilter" condition="status = :P1" />
                                        ↳ column name
  </class>
  <filter-def name="myfilter">
    <filter-param name="P1" type="string" />
  </filter-def>
</hibernate-mapping>
```

2) With annotations,

```
@Entity
@Table (name = "flight")
@FilterDef (name = "myfilter", parameters = @ParamDef(name = "p1", type = "string"))
@Filter (name = "myfilter", condition = "status = :P1")
public class flight
{
  ---
  ---
}
```

⇒ Before executing a query, if we want to attach condition also to the query then we need to enable filter by calling session.enableFilter().

⇒ If we want to execute a query with out condition then disableFilter by calling session.disableFilter().

> create table flight (fno varchar2(6) primary key, source varchar2(15), destination varchar2(15), status varchar2(15));

> Table created.

> insert into flight values ('0909', 'chennai', 'Hyderabad', 'delayed');

> 1 row inserted.

→ Like this insert records with all status to flight table.

Example

FilterApp

- src
  - default package
    - flight.java
    - Main.java
  - hibernate.cfg.xml

Flight.java

```
@Entity
@FilterDef( name="myfilter", parameters=@ParamDef( name="p1", type="string"))
@Filter (name="myfilter", condition = "status = :p1")
public class flight
{
    @Id
    @Column (name="fno")
    private string flightNO;
    private String source;
    private String destination;
    @Column( name="status")
    private String flightStatus;

    // setters and getters

}
```

Main.java.

```
public class Main
{
    public static void main (String args[])
    {
        SessionFactory factory = new Configuration()
            Session session = factory.openSession();
        Query query = session.createQuery("from Flight f");
        // enable filter
        Filter f = session.enableFilter("myfilter");
            f.setParameter ("p1", "Delayed");
        List list = query.list();
        Iterator it = list.Iterator();
        while (it.hasNext())
        {
            Flight flight = (Flight) it.next();
        S.o.P ( flight.getFlightNo() + "  " + flight.getSource() + "  " + flight.get
        }                Destination() + "  " + flight.getFlightStatus() );
```

```
S.O.P("=====");
//disable filter
session.disableFilter("myfilter");
List list2 = query.list();
Iterator it2 = list2.iterator();
while(it2.hasNext())
{
    Flight ft = (Flight) it2.next();
    S.O.P(ft.getFlightNO() + "  " + ft.getSource() + "  " + ft.getDestination()
                                                    + "  " + ft.getFlightStatus());
}
session.close();
factory.close();
}
}
```

⇒ Before executing the above application, we need to prepare, a table with name flight with some sample rows like the following.

Flight (table)

FNO	source	destination	status
F201	Hyd	Delhi	Delayed
Q909	chennai	cochin	Cancelled
R102	Bangalore	Tirupati	Ready

<u>o/p</u>

Batch Processing in hibernate

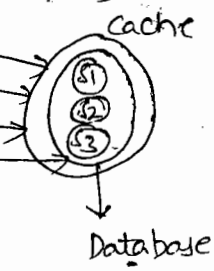
→ soultion `session.setFlushMode(FlushMode.MANUAL);`

```
Transaction tx = session.beginTransaction();
for (i=0; i<=10000; i++)
{
    session.save(ob);
    if(i%50==0)
    {
        session.flush();
        session.clear();
    }
}
tx.commit();
```

⇒ By default, hibernate will save objects in cache and when a transaction is committed then hibernate transfers objects from cache to database.

⇒ for example.

```
Transaction tx = session.beginTransaction();
session.save(S1);
session.save(S2);
session.save(S3);
tx.commit();
```



⇒ When a huge no.of objects are going to be saved in a cache then at some point of time we will get out of memory error.

⇒ To solve the problem, we use batch processing of hibernate.

⇒ In batch processing, we manually flush the cache for every specific no.of objects. so that we can avoid out of memory error.

⇒ The above solution code, we are flushing a cache for every 50 objects and then we are clearing the cache, to store the next objects.

Transaction Management (Tx Mgt)

⇒ The process of combining related operations into a single unit and executing them by applying do-everything or nothing principles is called Transaction Management.

⇒ We execute sensitive logics like ticket booking, transfer money & etc.. logics by enabling Transaction Management support.

⇒ TransferMoney operation is the combination of 2 operations

a) Withdraw amount from sauce amount

b) Deposit amount into another account

NOTE If one operations fails another operation should ~~do~~ be failed. This demands Transaction Management support

⇒ While working with Transaction Management 3 operations are important

a) Begin Transaction

b) Continue transaction

c) Commit or rollback transaction

⇒ If only resource (DB sw) is involved in Transaction Management operation then it is called Local Transaction

Ex: Transfer Money operation b/w two accounts of same bank

⇒ If more than resource (DB sw) is involved in Transaction Management operation then it is called Global/Distributed Transaction.

Ex: HB doesn't supports Global Transaction but spring, EJB supports Global Transactions

All technologies, fw supports Local Transactions (jdbc, hibernate, spring, ejb, --).

Sample Code in HB.

```
Transaction tx=null;
try{
    tx=ses.beginTransaction();
    With draw operation;
    Deposit operation;
   tx.commit();
}
catch(Exception e){
    tx.rollback();
}
```

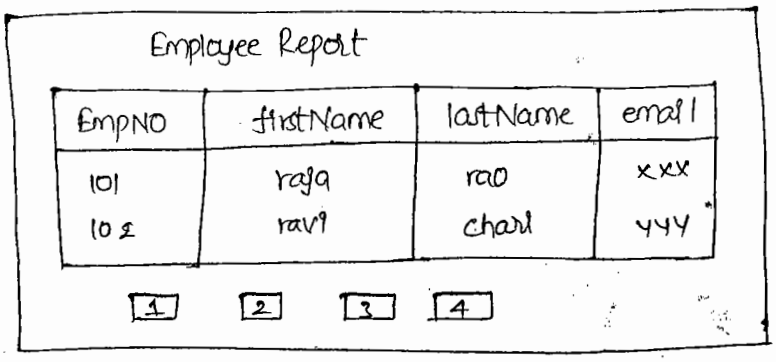
Pagination (using servlet with HB)

⇒ The process of displaying huge amount of records part by part in multiple pages called "Pageration". This is very useful in Report generation.

⇒ we generally do Report Generation by using Criteria API with the suppot of setFirstResult, setMaxResults Method as shown below.

```
Criteria ct = ses.createCriteria(EmpDetails.class);
         ct.setFirstResult(0);
         ct.setMaxResults(1);
        // execute QBC
List<EmpDetails> list = ct.list(); //gives 1st 3 records
        // process the results
for(EmpDetails ed: list)
 {
    s.o.p(ed);
 }
```

WebPage



HB Proj60 (Pagination)(ECDW)

```
|-> java resources
    |-> com.s.h.cfgs
            |-> hibernate.cfg.xml
    |-> com.s.h.domain |--> EmpDetails.java
    |-> com.s.h.dao |-> EmpDao.java, EmpDaoFactory.java, EmpDaoImpl.java
    |-> com.s.h.utility |-> HBUtil.java
    |-> com.s.h.test |-> ClientApp.java
|-> webcontent
    |-> index.html
    |-> WEB-INF |-> web.xml
```

Locking in Hibernate

⇒ If multiple client apps, Threads access the records of DB Table simultaneously or concurrently then it is possibility of getting data corruption.

⇒ To solve this concurrency problem we need to use locking concepts of Hibernate.

a) Optimistic Locking

⇒ Use versioning feature of hibernate for this locking

⇒ This locking throws exception if the version of the objects i.e., there with client App is not matching with the version of record in there with DB table

⇒ That exception will be raised is StaleObjectStateException / DirtyStateException i.e., state of the object i.e, there client App is not matching with the state of the record i.e, there is DB Table.

b) Pessimistic Locking

1) In this if one client access the record then lock will be applied on the record. So other clients can not access the records simultaneously until that lock is released.

2) For this we need to use ses.get() by specifying the lockMode as UPGRADE_NOWAIT

3) Pessimistic lock is recommended to use

4) Here No versioning is required

⇒ If we execute any application by enabling versioning features the optimistic lock will be applied automatically. If any version clash between client Application & DB table record StaleObjectStateException will be raised (refer APP31 of booklet).

Execution Procedure

a) Client1

Access the 101 record where version is 0 and goes to sleep state

b) Client2

Access the same 101 record and updates the record due to this version will be changed to 1 in DB table.

c) Client1

Completes the sleep state & come back to manipulate the record & notices versions are not matching due to this exception (StaleStateException) will be raised

Ex: App on Pessimistic Locking refer App32 of Main Booklet.

Execution

client1

Access the 101 record and gets into sleep state

(Record is accessed through ses.get(Product.class, 101, LockMode.UPGRADE_NOWAIT)

client2

During sleep period second client wants to access the 101 record. Due to this Error will be generated with message "Resource Busy".

Q Explain Different ORM Levels ? or ORM Quality Levels ?

Ref page no : 302, 303 of Main Booklet

Pure Relational (Stored Procedure)

Light Object Mapping (JDBC)

Medium Object Mapping (EJB)

Full Object Mapping (Composition, Inheritance, ..)

Q Mismatches b/w RDBMS DB s/w & Java Programming and how HB solves those mismatches

Inheritance (subtype Problems)

Java classes can be there in the inheritance, But table can't be placed in the inheritance for this use Inheritance Mapping concepts.

Problem of Granularity

Java classes can be designed through composition but DB Tables cannot be placed in Composition. HB gives component Mapping to solve this problem.

Identity problem

In Java we can check the quality either through "==" operators (or) .equals() method. In DB s/w record is identified with its PK column value. HB solves the problem by identifying the object through identity field cfg <id> (or) @Id.

Navigation Problem

In db tables we can navigate from one table record to another table record through FK colum. But navigation b/w Java objects is possible through reference variables (parent class should have child class reference & vice versa)

HB solves this problem through Association Mapping.