

Hibernate

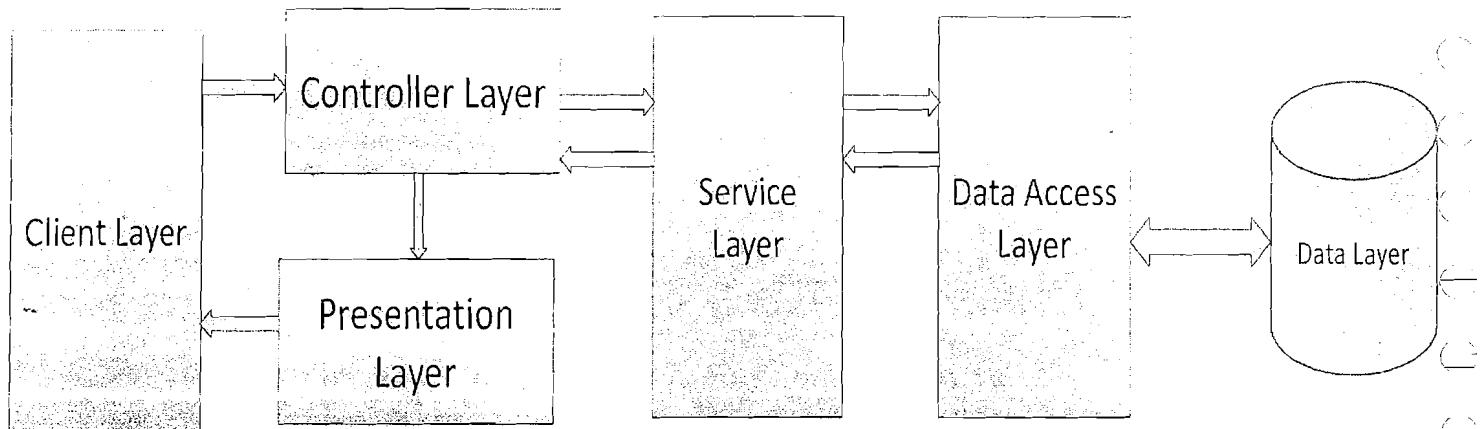
Q) Where actually java is used?

- ⇒ Java is used to develop the enterprise applications
- ⇒ Enterprise means business organization
- ⇒ Business organization provides services

Q) What is Enterprise Application?

- ⇒ Computerizing business services

Architecture of Enterprise Application



Client Layer:

- It is browser software.

Controller Layer:

- Receiving user request from client (calling request)
- Capturing the user provided data
- Validating the user input
- Calls the business method to get business services and get processed data
- Keep the processed data in memory(request/session/application scope)
- Finally forwarding the request to VIEW.

Note: should not write business-Logic/Data-Access-Logic in the Controller Layer. Because that is not reusable

Presentation Layer:

- Receive the control from Controller Layer
- Generate the output by taking the data from memory(request/session/application scope), which is stored by controller layer
- Generated output will be given to web-server, which intern return present the output to browser

Business/Service Layer:

- Receiving request from Controller Layer
- Contacting the Data Access Layer to get the database data
- Implementing the business logic
- Return the control/processed-data to Controller Layer

Persistent/Data-Access Layer:

- Receiving the request from business Layer
- Contacting database to get the database data
- Return the accessed data to business Layer

Data Layer:

- It is a database.

Q) What are the different logics available in Enterprise Application?

- **Presentation Logic:** Logic used to present the output/input.
- **Application/Controlling Logic:** Logic used to control the flow of application.
- **Business Logic:** Programmatical implementation of business rules is nothing but business logic.
- **Data Access Logic:** Logic used to contact the Database.

Q) What are the Sun Microsystems technologies and frameworks in enterprise application development?

Controller Layer	Presentation Layer	Business Layer	Data Access Layer
Servlets JSF	JSP	EJB2 session beans EJB3 session beans MDB(Message Driven Beans) WEB-SERVICES	JDBC Ejb2 entity beans Ejb3 entitys[JPA] (java persistence API)

Q) What are the non-Sun Microsystems technologies and frameworks in enterprise application development?

Controller Layer	Presentation Layer	Business Layer	Data Access Layer
Struts	HTML	Spring AOP	Hibernate
Spring Web MVC	Velocity	Spring JEE	Ibatis
Wicket	Freemarker	WEB-SERVICES	Toplink
Tapestry	Flex		JDO
Flash			Spring DAO
			Spring ORM

Objective of Hibernate: Developing Data access layer of an Enterprise application

Q) What is persistence in a java based enterprise application?

- The process of storing enterprise data in to relational database is known as persistence

Q) What is the traditional style of achieving persistence in java based enterprise application?

- Sending SQL statements to the Database using **JDBC API**

Q.) What are the limitations of the traditional approach?

- **Application portability** to the Database is lost (Vendor lock: diff SQL statement for the db's)
- **Mismatches** between Object oriented data representation and relational data representation are not properly addressed
- Requires the extensive knowledge of **DB**
- Manual operations on **Resultset**
- For every problem while communicating with the database (using JDBC), it throws same exception(**java.sql.SQLException**). As SQLException is **checked exception**, so we must write code in try-catch block or throws has to be specified.
- Need to implement **caching manually**
- In the Enterprise applications, the data flow with in an application from class to class will be in the **form of objects**, but while storing data finally in a database using JDBC then that object will be converted into **text**. Because JDBC doesn't **transfer objects** directly.

Q.) what is an alternative for traditional approach?

- **ORM** (Object Relational mapping)
- It is technique of **mapping** objected oriented data to that of relational data
- Through ORM technique **persistence services** (database) are provided to business layer in **pure object oriented** manner by overcoming all limitations of the traditional approach

Q.) What is Hibernate?

- Hibernate is an **ORM implementation**
- Hibernate is an Open source
- Hibernate is a framework
- Hibernate invented by **Gavin King**. He also invented **JBoss** server and **JPA**

- Hibernate is a **non-invasive** framework, means it won't forces the programmers to extend/implement any class/interface, and in hibernate we have all POJO classes so its light weight
- Hibernate can run **with or without server**, I mean it will suitable for all types of applications (desktop or web applications)

Q.) What is a framework?

- A framework is reusable semi finished application that can be customized to develop a specific application.

Q.) What are the features of hibernate?

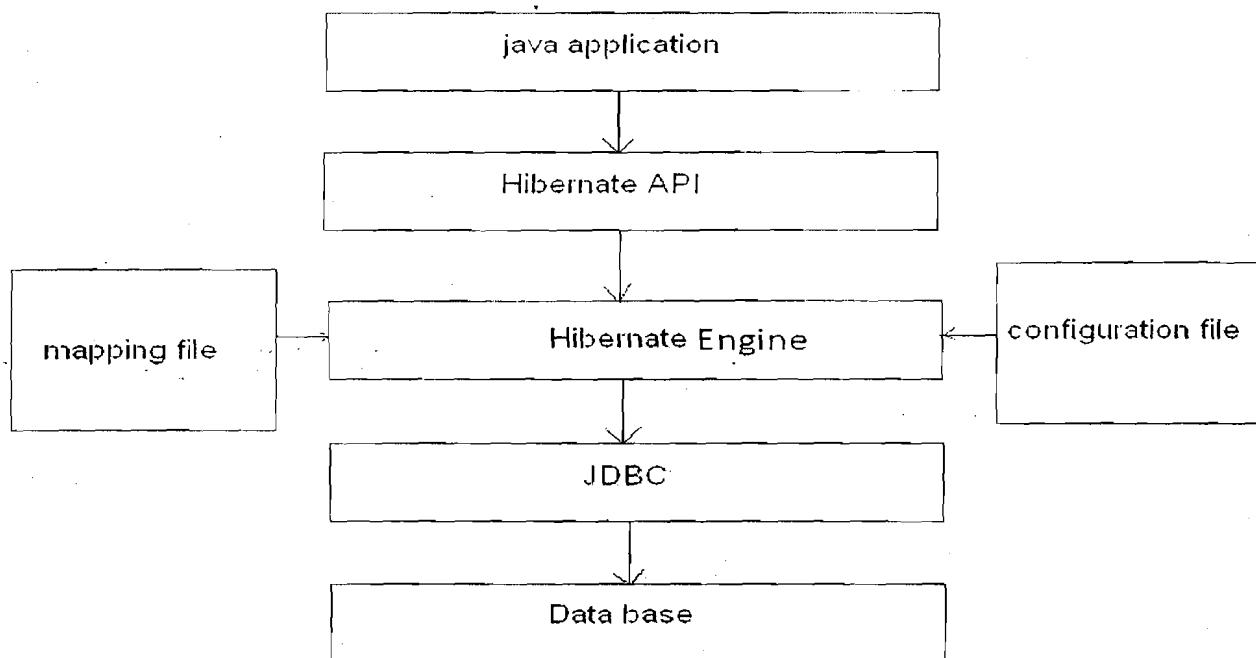
- Hibernate persists **java objects** into database (Instead of primitives)
- It provides Database services in **Database vendor independent** Manner, so that java applications become portable across the multiple databases
- Hibernate generates **efficient queries** for java application to communicate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

- Hibernate provides **transactional capabilities** that can work with both stand-alone or java Transaction API (JTA) implementations ...etc
- Hibernate supports **annotations**, apart from **XML**

Q.)What are the disadvantages of hibernate?

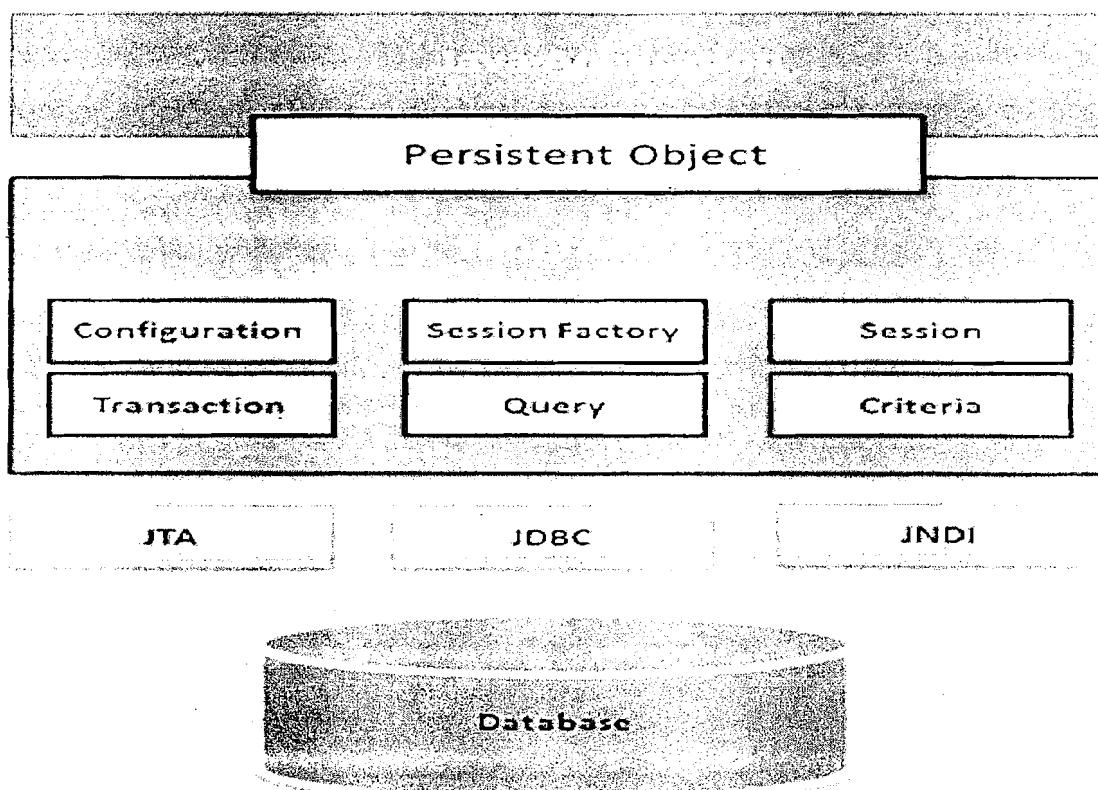
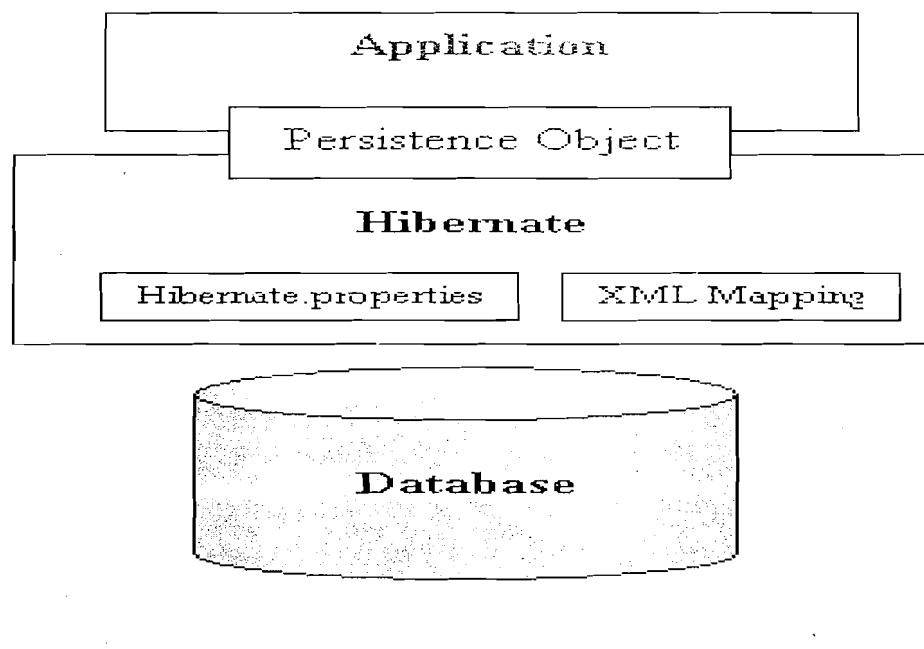
- Since hibernate generates lots of SQL statements at runtime so it is **slower than pure JDBC**
- Hibernate is not much flexible in case of **composite mapping**. This is not disadvantage since understanding of composite mapping is complex
- Hibernate does not support **some type of queries** which are supported by JDBC
- **Boilerplate code issue**, actually we need to write same code in several files in the same application, but **spring** eliminated this

Hibernate Architecture



- **Java Application** makes use of **hibernate API** methods calls to inform the persistent needs to hibernate. Then **Hibernate engine** generate **JDBC** code that corresponds to the underlying DB by using **mapping file** and **configuration file** information.
- We can also find the **architecture diagrams** as follows ...

Hibernate Architecture



```
2.  
3.   <class name="Entity class name" table="table name in database">  
4.     <id name="id variable name" column="primary column name in database" />  
5.     <property name="variable1 name" column="column name in database" />  
6.     <property name="variable2 name" column="column name in database" />  
7.   </class>  
8.  
9. </hibernate-mapping>
```

Syntax Of Mapping Annotations:

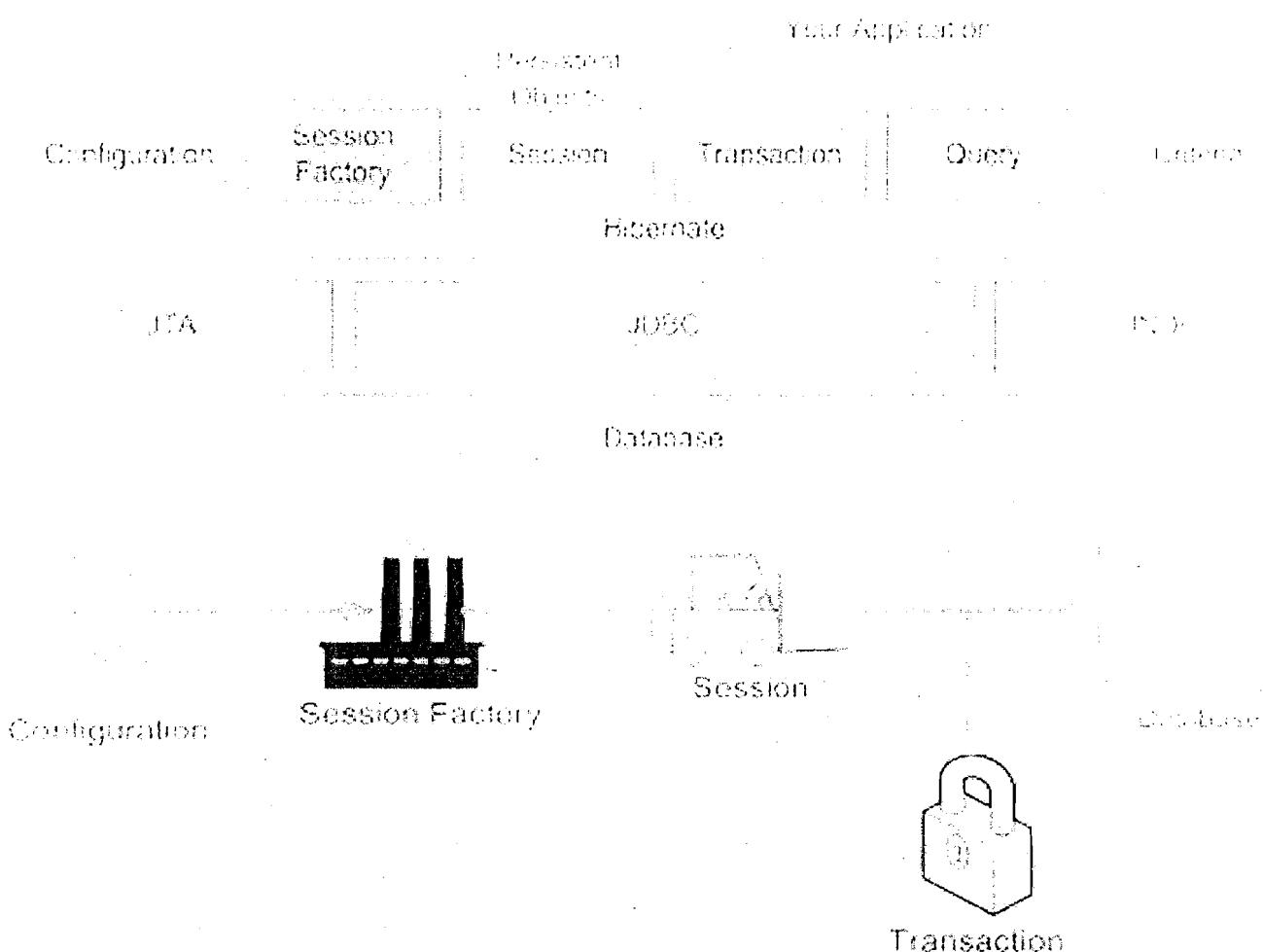
```
1. @Entity  
2. @Table(name = "table name in database")  
3. public class EntityName {  
4.     @Id  
5.     @Column(name = "primary column name in database")  
6.     private int idVariableName;  
7.  
8.     @Column(name = "column name in database ")  
9.     private String variableName1;  
10.    @Column(name = "column name in database ")  
11.    private String variableName2;  
12.    // setters & getters  
13. }
```

Q.) What is hibernate configuration file?

- It is an **XML** file in which **database connection details** (username, password, url, driver class name) and **Hibernate Properties**(dialect, show_sql, second_level_cache...etc) and **Mapping file name(s)** are specified to the hibernate
- Hibernate uses this file to establish connection to the particular database server
- Standard for this file is **<hibernate.cfg.xml>**
- We must create one **configuration** file for each **database** we are going to use, suppose if we want to connect with 2 databases, like **Oracle, MySql**, then we must create 2 configuration files.
No. of databases we are using = That many number of configuration files
- We can write this configuration in 2 ways...
 - **XML file**
 - **Properties file(old style)**
- We don't have annotations to write configuration details. Actually in hibernate 1.x, 2.x we defined this configuration by using **.properties** file, but from 3.x XML came into picture. XML files are always recommended to use.

Syntax Of Configuration xml:

```
1. <hibernate-configuration>  
2. <session-factory>  
3.  
4. <!-- Related to the connection START -->
```



Q.) What is hibernate mapping file?

- In this file hibernate application developer specify the mapping from **entity** class name to **table** name and entity **properties** names to table **column** names. i.e. mapping object oriented data to relational data is done in this file
- Standard name for this file is **<domain-object-name.hbm.xml>**
- In general, for each domain object we create one mapping file
Number of Entity classes = that many number of mapping xmls
- Mapping can be done using **annotations** also. If we use annotations for mapping then we no need to write mapping file.
- From hibernate 3.x version onwards it provides support for annotation, So mapping can be done in two ways
 - **XML**
 - **Annotations**

Syntax Of Mapping xml:

1. **<hibernate-mapping>**

```
5. <property name="connection.driver_class">Driver Class Name </property>
6. <property name="connection.url">URL </property>
7. <property name="connection.user">user </property>
8. <property name="connection.password">password</property>
9. <!-- Related to the connection END -->
10.
11. <!-- Related to hibernate properties START -->
12. <property name="show_sql">true/false</property>
13. <property name="dialet">Database dialet class</property>
14. <property name="hbm2ddl.auto">create/update or what ever</property>
15. <!-- Related to hibernate properties END-->
16.
17. <!-- Related to mapping START-->
18. <mapping resource="hbm file 1 name .xml" / >
19. <mapping resource="hbm file 2 name .xml" / >
20. <!-- Related to the mapping END -->
21.
22. </session-factory>
23. </hibernate-configuration>
```

Q.) What are the Simple Hibernate Application Requirements?

1. Entity class
2. Mapping file(Required if you are not using annotations)
3. Configuration file
4. DAO class (Where we write our logic to work with database)

Setting hibernate environment

- To work with hibernate framework we need to **add .jar(s) files** provided by that framework to our java application.
- No framework is installable software, it means we doesn't contain any setup.exe
- When we download any framework, we will get a '**zip**' file and we need to **unzip** it, to get the required jar files, actually all frameworks will follow same common principles like...
 - **Framework** will be in the form of a set of jar files, where one jar file acts as main (We can call this file as **core**) and remaining will acts as **dependent** jar files.
 - Each Framework contain at least one **configuration xml file**, but multiple configuration files also allowed.
- We can download hibernate jar files from the following links. Based on requirement we can download the corresponding version.
 - For version 2.x (<http://sourceforge.net/projects/hibernate/files/hibernate2/>)
 - For version 3.x (<http://sourceforge.net/projects/hibernate/files/hibernate3/>)
 - For version 4.x (<http://sourceforge.net/projects/hibernate/files/hibernate4/>)
- While downloading select **.zip** file for windows environment, select **.tar** file for unix environment.

- After downloading the zip file, unzip it and we can find the required jars in the extracted folder. If we consider 4.x version, we need the following jars to work with hibernate application.

```
• antlr-2.7.7.jar  
• dom4j-1.6.1.jar  
• hibernate-commons-annotations-4.0.1.Final.jar  
• hibernate-core-4.1.4.Final.jar  
• hibernate-entitymanager-4.1.4.Final.jar  
• hibernate-jpa-2.0-api-1.0.1.Final.jar  
• javassist-3.15.0-GA.jar  
• jboss-logging-3.1.0.GA.jar  
• jboss-transaction-api_1.1_spec-1.0.0.Final.jar
```

NOTE: Along with the hibernate jars we must include one more jar file, which is nothing but related to our database, this is depending on your database. For example, if we are working with Oracle we need to add **ojdbc6.jar**.

Q.) What are the Steps to develop hibernate applications?

Step 1: Develop persistent/domain/entity class for each table of the relational model

Step 2: For each entity develop a mapping file

Step 3: Develop the configuration file

Step 4: Add hibernate framework jar files in the classpath

Step 5: Make use of hibernate API and perform persistent operations

Q.) How to Make use of hibernate API to perform persistent operations?

STEP1: Create Configuration object

```
Configuration configuration = new Configuration();
```

STEP2: Read configuration file along with mapping files using configure() method of Configuration Object

```
configuration.configure();
```

STEP3: Build a SessionFactory from Configuration

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

STEP4: Get Session from SessionFactory object

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

STEP5: Perform persistence operations

The **Session** interface provides methods to perform **CRUD** (Create Read Update Delete) operations on the instances of mapped **entity** classes. Perform transactions if require while performing DML operations. Session interface methods are...

session save(s)	- Inserting object 's' into database
session update(s)	- Updating object 's' in the database
session load(s)	- Selecting object 's' object
session delete(s)	- Deleting object 's' from database

STEP6: Close the session

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

Final flow will be...

```
Configuration  
SessionFactory  
Session  
Transaction  
Close Statements
```

Q.) Develop Hibernate application, in which we can perform account creation, retrieve, update and delete?

SQL Script

```
CREATE TABLE ACCOUNT (
    ACCNO      NUMBER (5) NOT NULL,
    NAME       VARCHAR2 (20) NOT NULL,
    BAL        NUMBER(8,2) NOT NULL,
    CREATION_DT DATE      NOT NULL,
    PRIMARY KEY ( ACCNO )
)
```

ACCOUNT TABLE

ACCNO	NAME	BAL	CREATION_DT
90001	kesavareddy	69541.03	5/29/2012
90002	yellareddy	86568.69	5/15/2012
90003	cherry	56489.26	5/2/2012



hibernate.cfg.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3.           "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.           "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5.
6. <hibernate-configuration>
7.
8.   <session-factory>
9.     <property name="dialect">
10.      org.hibernate.dialect.Oracle9Dialect
11.    </property>
12.    <property name="connection.url">
13.      jdbc:oracle:thin:@localhost:1521:XE
```

Hibernate-CRUD Application

By Mr. SekharReddy

```
14.      </property>
15.      <property name="connection.username">system</property>
16.      <property name="connection.password">tiger</property>
17.      <property name="connection.driver_class">
18.          oracle.jdbc.driver.OracleDriver
19.      </property>
20.      <property name="connection.pool_size">15</property>
21.      <property name="show_sql">true</property>
22.      <property name="hibernate.auto">update</property>
23.      <property name="use_sql_comments">true</property>
24.      <property name="format_sql">true</property>
25.      <mapping resource="com/sekharit/hibernate/mapping/Account.hbm.xml" />
26.  </session-factory>
27.
28. </hibernate-configuration>
29.
```

SessionUtil.java

```
1. package com.sekharit.hibernate.util;
2.
3. import org.hibernate.Session;
4. import org.hibernate.cfg.Configuration;
5.
6. public class SessionUtil {
7.
8.     private static final ThreadLocal<Session> threadLocal = new ThreadLocal<Session>();
9.     private static org.hibernate.SessionFactory sessionFactory;
10.
11.    static {
12.        try {
13.            sessionFactory = new Configuration().configure(
14.                "com/sekharit/hibernate/config/hibernate.cfg.xml").buildSessionFactory();
15.        } catch (Exception e) {
16.            e.printStackTrace();
17.        }
18.    }
19.
20.    private SessionUtil() {
21.    }
22.
23.    public static Session getThreadLocalSession() {
24.        Session session = (Session) threadLocal.get();
25.
26.        if (session == null) {
27.            session = sessionFactory.openSession();
28.            threadLocal.set(session);
29.        }
30.
31.        return session;
32.    }
33.
34.    public static void closeThreadLocalSession() {
```

```
35.         Session session = (Session) threadLocal.get();
36.         threadLocal.set(null);
37.
38.         if (session != null) {
39.             session.close();
40.         }
41.     }
42.
43.     public static Session getSession() {
44.         return sessionFactory.openSession();
45.     }
46.
47.     public static void closeSession(Session session) {
48.         if (session != null) {
49.             session.close();
50.         }
51.     }
52.
53. }
```

Account.java

```
1. package com.sekharit.hibernate.bean;
2.
3. import java.util.Date;
4.
5. public class Account {
6.     private long accno;
7.     private String name;
8.     private double balance;
9.     private Date creationDate;
10.
11.    public long getAccno() {
12.        return accno;
13.    }
14.
15.    public void setAccno(long accno) {
16.        this.accno = accno;
17.    }
18.
19.    public String getName() {
20.        return name;
21.    }
22.
23.    public void setName(String name) {
24.        this.name = name;
25.    }
26.
27.    public double getBalance() {
28.        return balance;
29.    }
}
```

Hibernate-CRUD Application

By Mr. SekharReddy

```
30.  
31.     public void setBalance(double balance) {  
32.         this.balance = balance;  
33.     }  
34.  
35.     public Date getCreationDate() {  
36.         return creationDate;  
37.     }  
38.  
39.     public void setCreationDate(Date creationDate) {  
40.         this.creationDate = creationDate;  
41.     }  
42.  
43.     @Override  
44.     public String toString() {  
45.         return "Account [accno=" + accno + ", balance=" + balance  
46.                 + ", creationDate=" + creationDate + ", name=" + name + "]";  
47.     }  
48. }
```

Account.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>  
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  
4. <hibernate-mapping schema="SYSTEM" >  
5.     <class name="com.sekharit.hibernate.bean.Account" table="ACCOUNT">  
6.         <id name="accno" type="long" >  
7.             <column name="ACCNO" length="5" not-null="true" ></column>  
8.         </id>  
9.         <property name="name" type="string" >  
10.            <column name="NAME" length="20" not-null="true" ></column>  
11.        </property>  
12.        <property name="balance" type="double" >  
13.            <column name="BAL" precision="8" scale="2" not-null="true" ></column>  
14.        </property>  
15.        <property name="creationDate" type="date" >  
16.            <column name="CREATION_DT" not-null="true" ></column>  
17.        </property>  
18.    </class>  
19. </hibernate-mapping>
```

AccountDAO.java

```
1. package com.sekharit.hibernate.dao;  
2.  
3. import org.hibernate.HibernateException;  
4. import org.hibernate.Session;  
5. import org.hibernate.Transaction;  
6.  
7. import com.sekharit.hibernate.bean.Account;  
8. import com.sekharit.hibernate.util.SessionUtil;
```

```
9.  
10. public class AccountDAO {  
11.     public Account get(long accno) {  
12.         Session session = null;  
13.         Account account = null;  
14.         try {  
15.             session = SessionUtil.getSession();  
16.             account = (Account) session.get(Account.class, accno);  
17.         } catch (HibernateException e) {  
18.             e.printStackTrace();  
19.         } finally {  
20.             SessionUtil.closeSession(session);  
21.         }  
22.     }  
23.  
24.     return account;  
25. }  
26.  
27. public void insert(Account account) {  
28.  
29.     Session session = null;  
30.     try {  
31.         session = SessionUtil.getSession();  
32.         session.beginTransaction().begin();  
33.         session.save(account);  
34.         session.getTransaction().commit();  
35.     } catch (HibernateException e) {  
36.         session.getTransaction().rollback();  
37.         e.printStackTrace();  
38.     } finally {  
39.         SessionUtil.closeSession(session);  
40.     }  
41. }  
42.  
43. public void update(Account account) {  
44.     Session session = null;  
45.     try {  
46.         session = SessionUtil.getSession();  
47.         session.beginTransaction().begin();  
48.         session.update(account);  
49.         session.getTransaction().commit();  
50.     } catch (HibernateException e) {  
51.         session.getTransaction().rollback();  
52.         e.printStackTrace();  
53.     } finally {  
54.         SessionUtil.closeSession(session);  
55.     }  
56. }  
57.  
58. public void delete(long accno) {
```

```
59.     Session session = null;
60.     Transaction transaction = null;
61.     try {
62.         session = SessionUtil.getSession();
63.         transaction = session.beginTransaction();
64.         Account account = (Account) session.get(Account.class, accno);
65.         session.delete(account);
66.         transaction.commit();
67.     } catch (HibernateException e) {
68.         transaction.rollback();
69.         e.printStackTrace();
70.     } finally {
71.         SessionUtil.closeSession(session);
72.     }
73. }
74. }
```

AccountService.java

```
1. package com.sekharit.hibernate.service;
2.
3. import java.util.Date;
4.
5. import com.sekharit.hibernate.bean.Account;
6. import com.sekharit.hibernate.dao.AccountDAO;
7.
8. public class AccountService {
9.     public static void main(String[] args) {
10.         AccountDAO dao = new AccountDAO();
11.
12.         // Retrieve Account
13.         Account rAccount = dao.get(90001);
14.         System.out.println("Account details ....");
15.         System.out.println("Accno : " + rAccount.getAccno());
16.         System.out.println("Name : " + rAccount.getName());
17.         System.out.println("Balance : " + rAccount.getBalance());
18.         System.out.println("Creation Date: " + rAccount.getCreationDate());
19.
20.         // Create Account
21.         Account cAccount = new Account();
22.         cAccount.setAccno(90005);
23.         cAccount.setName("sekhar");
24.         cAccount.setBalance(6899);
25.         cAccount.setCreationDate(new Date());
26.         dao.insert(cAccount);
27.         System.out.println("Account created successfully");
28.
29.         // Update Account
30.         Account uAccount = new Account();
31.         uAccount.setAccno(90003);
32.         uAccount.setName("sekhareddy");
```

```
33.         uAccount.setBalance(4500);
34.         uAccount.setCreationDate(new Date());
35.         dao.update(uAccount);
36.         System.out.println("Account updated successfully");
37.
38.     // Delete Account
39.     dao.delete(90002);
40.     System.out.println("Account is deleted successfully");
41.
42. }
43. }
```

After Execution ACCOUNT TABLE :

ACCNO	NAME	BAL	CREATION_DT
90001	sekharreddy	69541.03	5/29/2012
90003	sekhareddy	4500	5/29/2012
90005	sekhar	6899	5/29/2012

Q.) Rewrite the above application, using annotations instead of mapping file?

```
hibernateCRUDannotation
  src
    com.sekharit.hibernate.bean
      Account.java
    com.sekharit.hibernate.config
      hibernate.cfg.xml
    com.sekharit.hibernate.dao
      AccountDAO.java
    com.sekharit.hibernate.service
      AccountService.java
    com.sekharit.hibernate.util
      SessionUtil.java
  JRE System Library [jre6]
  Hibernate 3.3 Annotations & Entity Manager
  Hibernate 3.3 Core Libraries
  Referenced Libraries
    ojdbc14.jar
  lib
```

hibernate.cfg.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3.   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.   "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5.
6. <hibernate-configuration>
7.
8.   <session-factory>
9.     <property... />
10.    .....
```

Hibernate-CRUD Application

By Mr. SekharReddy

```
23. }
24.
25. public void setAccno(long accno) {
26.     this.accno = accno;
27. }
28.
29. @Column(name = "NAME", nullable = false, length = 20)
30. public String getName() {
31.     return name;
32. }
33.
34. public void setName(String name) {
35.     this.name = name;
36. }
37.
38. @Column(name = "BAL", nullable = false, precision=8, scale=2)
39. public double getBalance() {
40.     return balance;
41. }
42.
43. public void setBalance(double balance) {
44.     this.balance = balance;
45. }
46.
47. @Temporal(TemporalType.DATE)
48. @Column(name = "CREATION_DT", nullable = false)
49. public Date getCreationDate() {
50.     return creationDate;
51. }
52.
53. public void setCreationDate(Date creationDate) {
54.     this.creationDate = creationDate;
55. }
56.
57. @Override
58. public String toString() {
59.     return "Account [accno=" + accno + ", balance=" + balance
60.             + ", creationDate=" + creationDate + ", name=" + name + "]";
61. }
62.
63. }
```

Account.hbm.xml

---NOT REQUIRED---

AccountDAO.java

---Same As Above---

AccountService.java

---Same As Above---

Hibernate – The type AnnotationConfiguration is deprecated

Problem

- ⇒ Working with Hibernate 3.6, noticed the previous “org.hibernate.cfg.AnnotationConfiguration”, is marked as “deprecated”.

Code snippets ...

```
import org.hibernate.cfg.AnnotationConfiguration;
//...
private static SessionFactory buildSessionFactory() {
    try {
        return new AnnotationConfiguration().configure().buildSessionFactory();

    } catch (Throwable ex) {

        System.err.println("Initial SessionFactory creation failed." + ex);
        throw new ExceptionInInitializerError(ex);
    }
}
```

- ⇒ The code is still working, just keep displaying the deprecated warning message, is there any replacement for “AnnotationConfiguration” ?

Solution

- ⇒ In Hibernate 3.6, “org.hibernate.cfg.AnnotationConfiguration” is deprecated, and all its functionality has been moved to “org.hibernate.cfg.Configuration”.
- ⇒ So , you can safely replace your “AnnotationConfiguration” with “Configuration” class.

Code snippets ...

```
import org.hibernate.cfg.Configuration;
//...
private static SessionFactory buildSessionFactory() {
    try {
        return new Configuration().configure().buildSessionFactory();

    } catch (Throwable ex) {

        System.err.println("Initial SessionFactory creation failed." + ex);
        throw new ExceptionInInitializerError(ex);
    }
}
```

Hibernate-CRUD Application

By Mr. SekharReddy

```
11. ....  
12. <mapping class="com.sekharit.hibernate.bean.Account" />  
13. </session-factory>  
14.  
15. </hibernate-configuration>  
16.
```

SessionUtil.java

```
1. package com.sekharit.hibernate.util;  
2. import org.hibernate.Session;  
3. import org.hibernate.cfg.AnnotationConfiguration;  
4. public class SessionUtil {  
5.  
6.     private static final ThreadLocal<Session> threadLocal = new ThreadLocal<Session>();  
7.     private static org.hibernate.SessionFactory sessionFactory;  
8.  
9.     static {  
10.         try {  
11.             sessionFactory = new AnnotationConfiguration().configure(  
12.                         "com/sekharit/hibernate/config/hibernate.cfg.xml").buildSessionFactory();  
13.         } catch (Exception e) {  
14.             e.printStackTrace();  
15.         }  
16.     }  
17.     .....  
18.     .....  
19. }
```

Account.java

```
1. package com.sekharit.hibernate.bean;  
2.  
3. import java.util.Date;  
4.  
5. import javax.persistence.Column;  
6. import javax.persistence.Entity;  
7. import javax.persistence.Id;  
8. import javax.persistence.Table;  
9. import javax.persistence.Temporal;  
10. import javax.persistence.TemporalType;  
11. @Entity  
12. @Table(name="ACCOUNT")  
13. public class Account {  
14.     private long accno;  
15.     private String name;  
16.     private double balance;  
17.     private Date creationDate;  
18.  
19.     @Id  
20.     @Column(name = "ACCNO", nullable = true, length=5)  
21.     public long getAccno() {  
22.         return accno;
```

Q.) Explain more about **hibernate.cfg.xml** (configuration file)?

- This xml file used to specify locations of mapping files/Entities
- In projects we don't give the database details(url, username, password, driverclass) in the configuration file, instead of that, we give JNDI name of DataSource.

```
<property name="connection.datasource">myDataSourceName</property>
```

- Hibernate supports default connection pooling but which will not be used in projects. We use always server provided connection pooling.

1) **connection.pool_size:** Used to configure hibernate provided connection pooling in hibernate.cfg.xml

```
<property name="connection.pool_size">10</property>
```

2) **show_sql:** if the 'show_sql' value is **true** we can view all the hibernate generated queries in the console.

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

3) **use_sql_comments :** To add SQL comment to the SQL query generated by Hibernate

```
<property name="use_sql_comments">true</property>
```

4) **format_sql :** Format the SQL query, so that easy to read

```
<property name format_sql">true</property>
```

5) **hbm2ddl.auto:** It has two values

- a. create or create-drop
- b. update

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

Case 1: table does not exist

Create new schema based on the mapping file configurations

Case 2: table exists

Drop the existing schema and create a new schema based on the mapping file configurations

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

Case 1 : table doesn't exist

Create a new schema based on the mapping file configurations

Case 2 : table exists

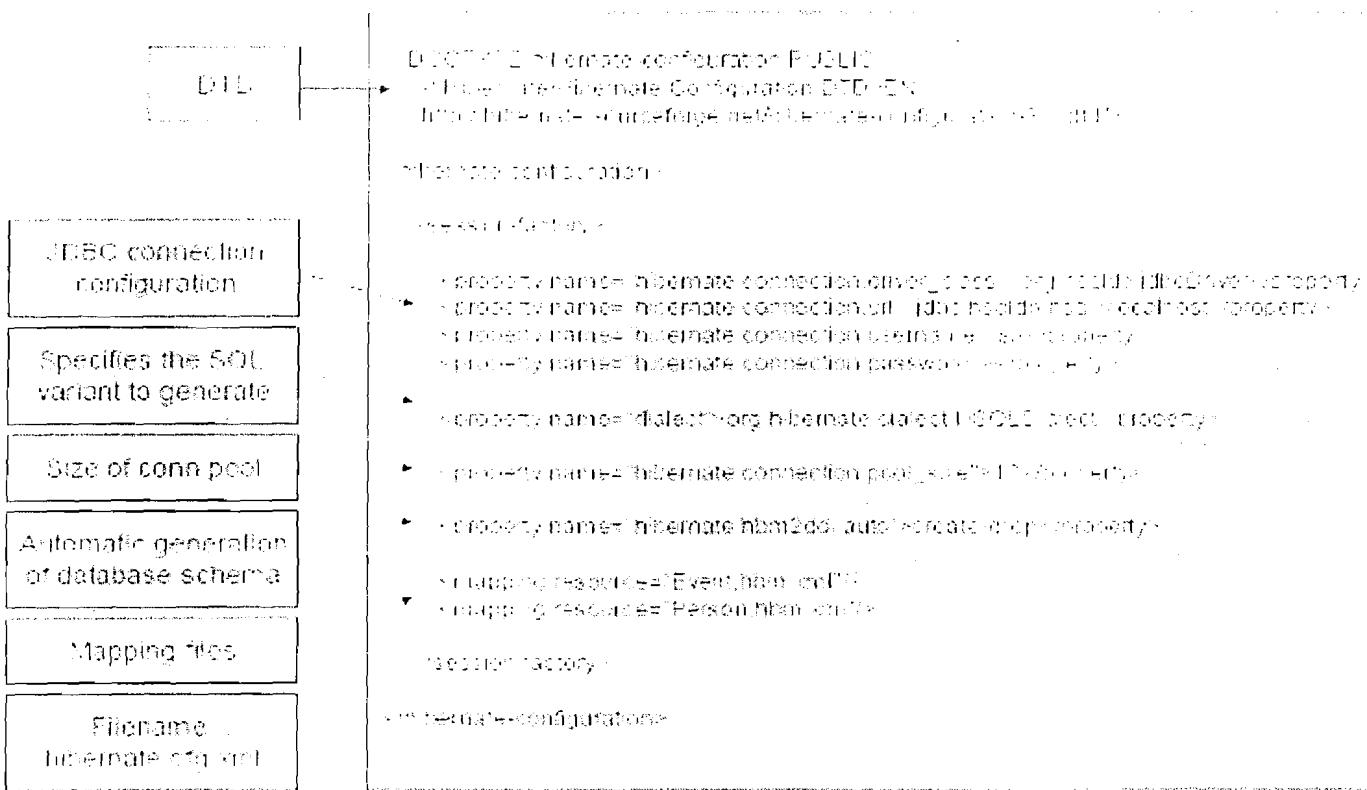
Use the existing schema

- If its value is 'create' while running the application hibernate will drop the old schema and it will create the new schema. (Based on HBM file)

Hibernate-Architectural Elements

Mr.SekharReddy

- If its value is ‘update’, hibernate will check for schema existence. If schema doesn’t exist it will create the new schema. If the schema already exists it uses the existing schema for persistence operations.



Q.) What do you know about **dialect** in Hibernate?

- Dialect class is a simple java class, which contains mapping between java language data type and database data type.
 - Dialect class contains queries format for predefined hibernate methods
 - Hibernate generates queries for the specific database based on the **Dialect** class. If you want to shift from one database to another just change the Dialect class name in **hibernate.cfg.xml** file.
 - All Dialect classes must extend ‘**Dialect**’ (abstract) class
 - Hibernate supports almost 30 dialect classes.
 - If we want we can write our own dialect by extending **Dialect** class
 - Dialect class is used convert **HQL** queries into database specific queries.

Hibernate SQL Dialects (`hibernate.dialect`)

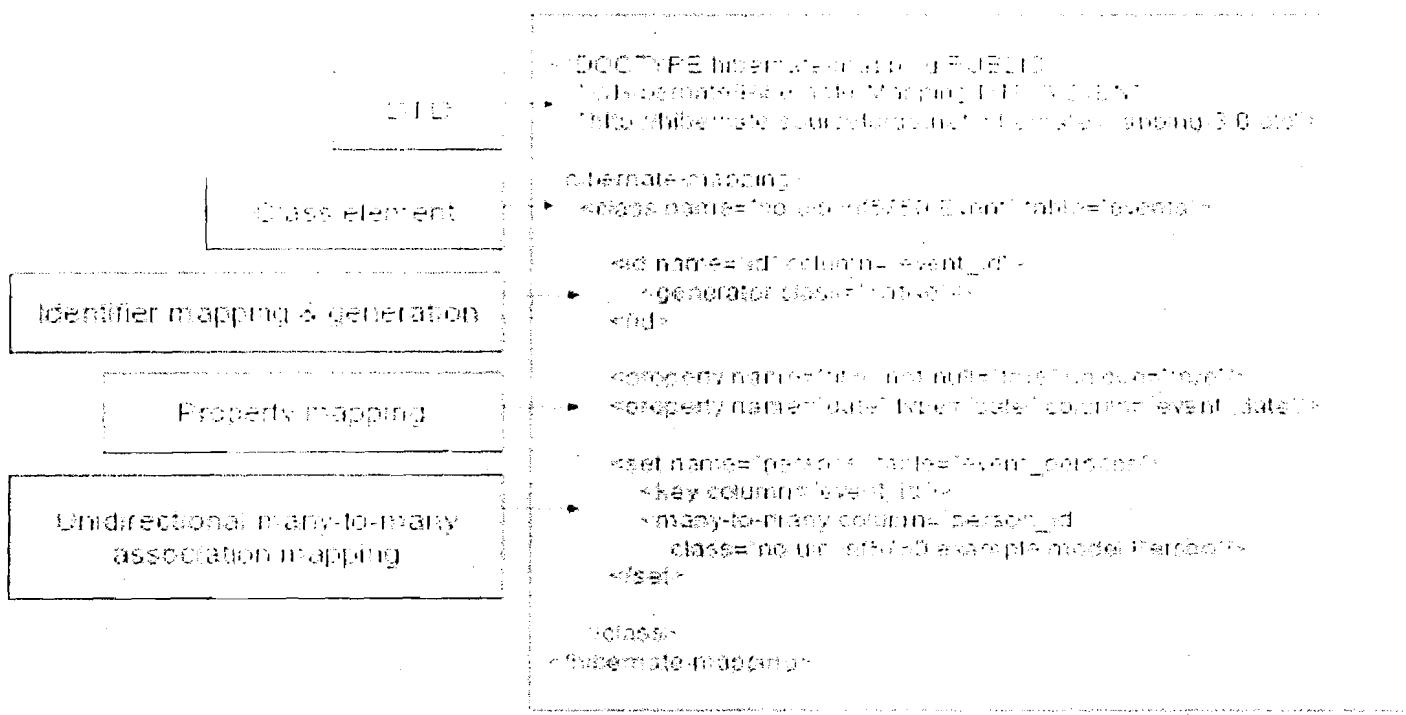


RDBMS	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQL5Dialect
MySQL with InnoDB	org.hibernate.dialect.MySQL5InnoDBialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version)	org.hibernate.dialect.Oracle8iDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
Firebird	org.hibernate.dialect.FirebirdDialect

Q.) Explain more about Hibernate mapping file?

- Each hibernate mapping file must contain only one `<id>` (or relevant tag `<composite-id>`)
- Java object identified uniquely by the `<id>` tag property.
- `<id>` tag property corresponding column can be **primary key** or **non-primary key** in the database.
- In mapping file class names and property names are **case sensitive**. But Table names and column names are not case sensitive.

- When the property name and column name both are same we no need to give 'column' attribute.
- When the Persistent class name and table name both are same we no need to give 'table' attribute.
- Generally we write one mapping file per one domain object. But it allows writing multiple objects mapping information within the same mapping file. Per each class mapping we need to write one <class> tag.



- Databases have different ways to organize its tables. Some database places all tables in a different "schemas", some database places all tables in a different "catalogs". If we want we can specify this in <class> tag of the mapping file
- In mapping file we write fully qualified name of the entity class in "name" attribute of <class> tag. Instead of that we can write package name separately using "package" attribute of <hibernate-mapping> tag
- In the mapping file we no need to map all the properties of the entity and all the columns of table. As per our requirement we configure required properties of the entity and columns of the table

Q.) Explain about annotations which are used in our application to map entity to table?

- In our examples if we observe, we used the following annotations
 - **@Entity**
 - **@Table**
 - **@Id**
 - **@Column**

- `@GeneratedValue`
 - `@Temporal`
- All the above annotations we are taking from `java.persistence` package. Actually this package is not the part of hibernate API. This package is from JPA(Java Persistence API).
- These annotations also given by hibernate. But we don't prefer to use hibernate given annotations. We prefer to use JPA annotations. Reason for this is, Hibernate is a specific API, where as JPA is a specification.
- If we use JPA annotations we have to flexibility to change the implementation vendor without changing application code.
- JPA is an API(from SUN), its not the implementation. There are multiple implementations are there for JPA. Some of famous implementation of JPA are **OpenJPA**, **Hibernate**, **Toplink Essentials**, **Eclipselink** ...etc.

Q.) What is Configuration object?

- Object Oriented representation of hibernate configuration file along with mapping file is known as **Configuration** object
- By default Hibernate reads configuration file with name "`hibernate.cfg.xml`" which is located in "classes" folder
- If we want to change the file name or if we want change the location of "`hibernate.cfg.xml`" then we need to pass user given configuration file name (along with path) to "`configure()`" method of **Configuration** class
- **Configuration** object Stores the configuration file data in different variables. Finally all these variables are grouped and create one high level hibernate object called **SessionFactory** object.
- So **Configuration** object only meant for creating **SessionFactory** object
- If we want we can provide the configuration information programmatically, without writing configuration file.(But it will become Hard coding, so not advisable)

Programmatic configuration

Adding mapping files to configuration object programmatically

```
Configuration cfg = new Configuration()  
    .addResource("Item.hbm.xml")  
    .addResource("Bid.hbm.xml");
```

Adding Entities(annotated persistent classes) to configuration object programmatically

```
Configuration cfg = new Configuration()  
.addClass(org.hibernate.auction.Item.class)  
.addClass(org.hibernate.auction.Bid.class);
```

To add hibernate properties to Configuration object programmatically

```
Configuration cfg = new Configuration()  
.addClass(org.hibernate.auction.Item.class)  
.addClass(org.hibernate.auction.Bid.class)  
.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBialect")  
.setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")  
.setProperty("hibernate.order_updates", "true");
```

This is not the only way to pass configuration properties to Hibernate. Some alternative options include:

1. Pass an instance of java.util.Properties to Configuration.setProperties().
2. Place a file named **hibernate.properties** in a root directory of the classpath.
3. Set System properties using java -Dproperty=value.
4. Include <property> elements in hibernate.cfg.xml (this is discussed later).

If you want to get started quickly **hibernate.properties** is the easiest approach.

The org.hibernate.cfg.Configuration is intended as a startup-time object that will be discarded once aSessionFactory is created.

Q.) What is SessionFactory?

- **SessionFactory** is an interface and **SessionFactoryImpl** is the implemented class
- It is factory of **Session** objects
- It is heavy weight object that has to be created only once per application. **SessionFactory** object provides lightweight **Session** objects.
- **SessionFactory** is not singleton. Lets create it only once using **Util/Helper** class
- **SessionFactory** is a Thread safe object.
- You need one **SessionFactory** object per database. So if you are using multiple databases then you would have to create multiple **SessionFactory** objects.
- **SessionFactory** is also responsible for second-level caching.

Q.) In one application, how many **SessionFactory** objects I can use ?

- A **SessionFactory** is pretty heavyweight, so, we recommend creating one and **caching** it in a singleton type of way. Then, you can create as many **Session** objects from it as you like.

Q.) Why **SessionFactory** is heavy weight?

- **SessionFactory** encapsulates **Session** objects, **Connections**, **Hibernate-properties**, **cashing** and **mappings**.

Q.) What do you know **Session** object?

- Session is an interface and SessionImpl is implemented class, both are given in org.hibernate.*;
- Session object is called **persistent manager** for the hibernate application.
- It is a single-threaded(not-thread safe), short-lived object
- It Wraps a JDBC connection
- The Hibernate **Session** operates using a single JDBC connection which can be injected by the hibernate while constructing session object.
- It has convenience methods to perform persistent operations.
- It is a factory for **Transaction** objects
- Holds a mandatory (**first-level**) cache of persistent objects

Note: After we complete the use of the Session, it has to be closed, to release all the resources such as associated objects and wrapped JDBC connection.

Q.) What is Transaction ?

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

Sample transaction code is as follows...

```
1.     Session = sessionFactory.openSession();
2.     Transaction tx = null;
3.     try {
4.         tx = session.beginTransaction();
5.         // DML operations
6.         tx.commit();
7.     } catch(Exception e) {
```

```

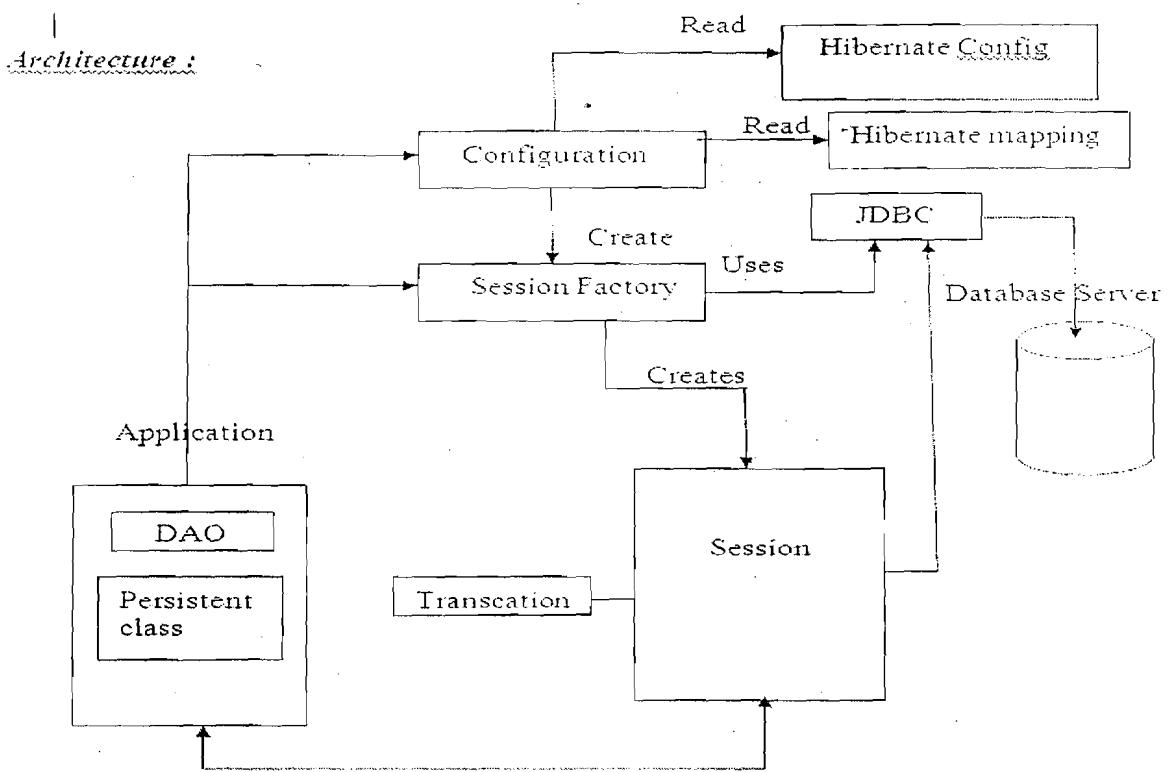
8.           tx.rollback();
9.       } finally{
10.           session.close();
11.       }
12.   }

```

Q.) Why we have written SessionUtil class?

- ⇒ When some common logic is repeating across the multiple classes of an application, it is better to move the common logic into some util class. Where ever we need that common logic we make use of util class.
- ⇒ **SessionFactory** is heavy weight and it's not singleton, so we should make one **SessionFactory** per database.
- ⇒ Using **Util /Helper** class we read the configuration file only once and we create one **SesionFactory** and we are providing facility to get session object and to close session object.

Hibernate complete architecture



configure a database driver in eclipse:

- Launch or open **MyEclipse Database Explorer** prospective. In the DB Browser window right click on the white place and select “new” option.
- When we choose the new option, it will launch database driver dialog box. Now we need to fill the following options in the dialog box.
- Select **driver template as oracle(thin driver)**, if we are using oracle database.
- Give the **driver name** (logical name, which is used to refer this configuration)

- Give the **connection url**.
- Give the **username** and **password**.
- Click on **Add Jars** button, then select classes.jar file
- Click on save password
- Click on finish

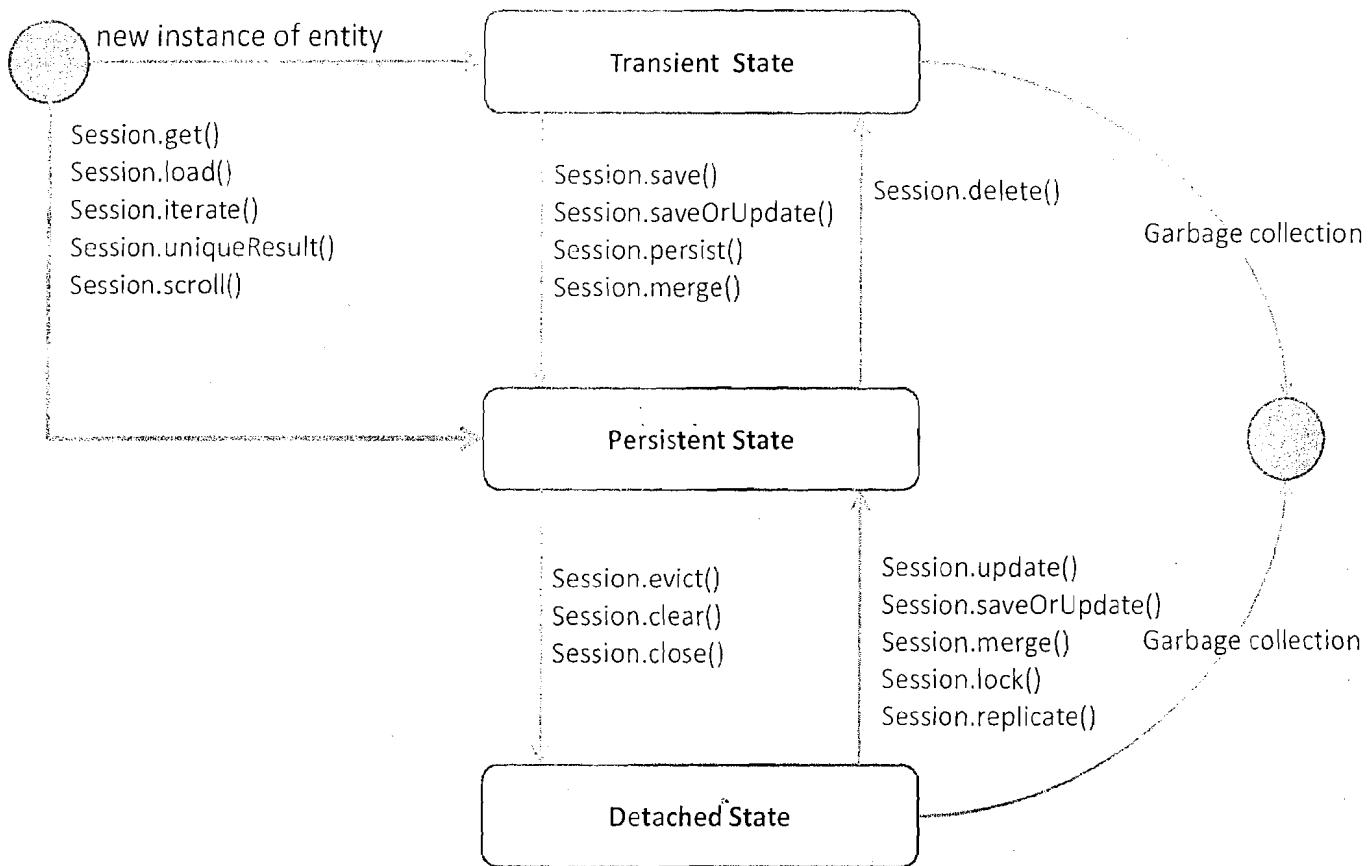
Hibernate Reverse engineering

- Open Myeclipse database explorer prospective.
- Select the database driver which we have created and right click on it and select open connection
- It will display a dialog box. Fill the required username and password.
- Select appropriate schema where our tables are stored
- Right click on the table and select “hibernate reverse engineering”
- The above operation will display a dialog box in that
- we need to select the “java source folder (src)”
- select the java package and click on finish

Persistent Object life cycle

Persistent Object has three life cycle states

1. Transient state
2. Persistent state
3. Detached state



Life cycle\object description	Object associate with session	Object present in the database
Transient state	NO	NO
Persistent state	YES	YES
Detached state	NO	YES

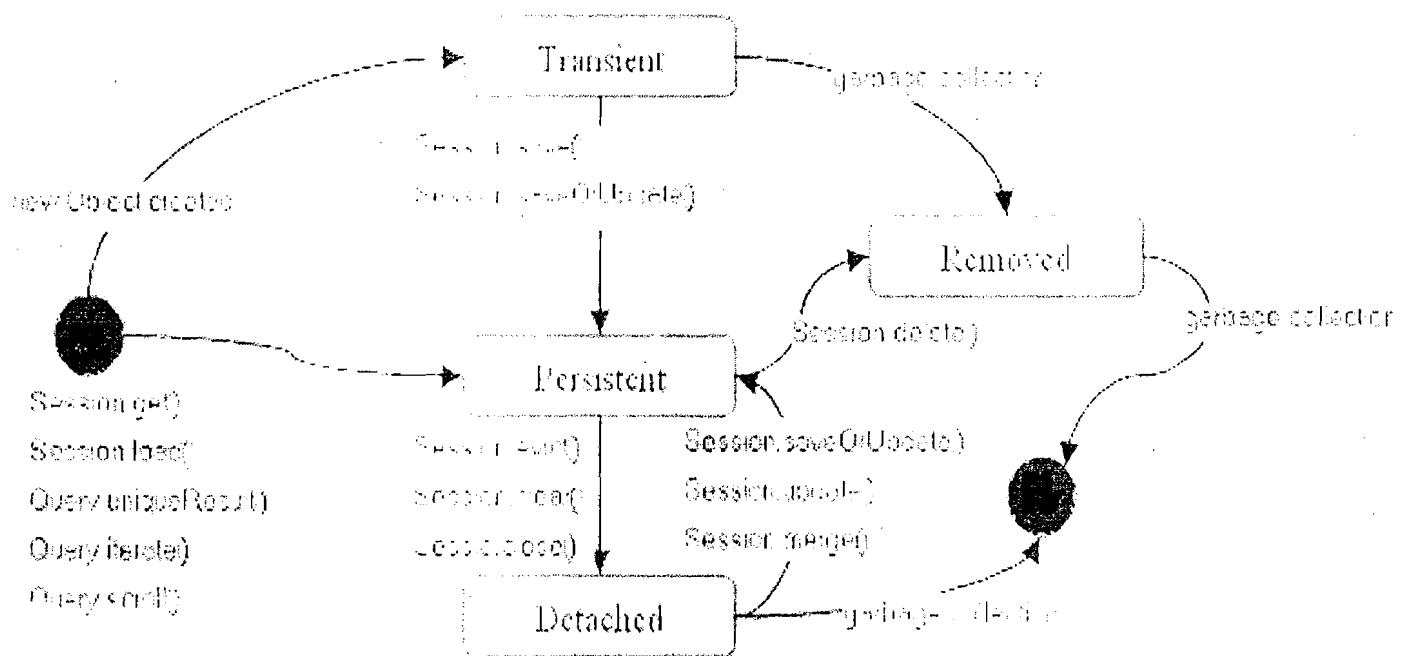
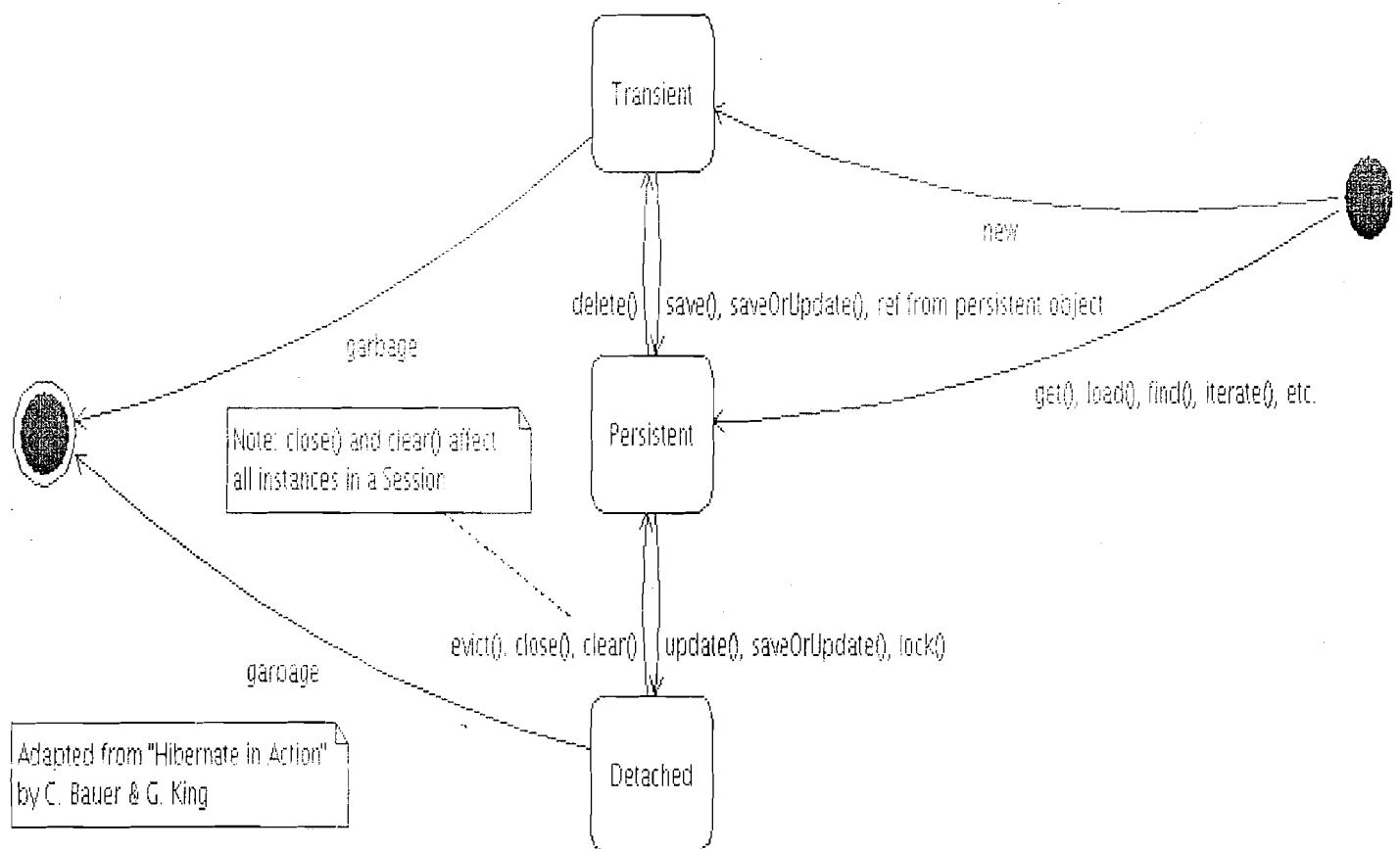
Note:

- **Object is associated with session** means object reference is available to the session object
- **Object present in the database** means object identifier value is available in database primary key column (Non identifier column values are not matter)

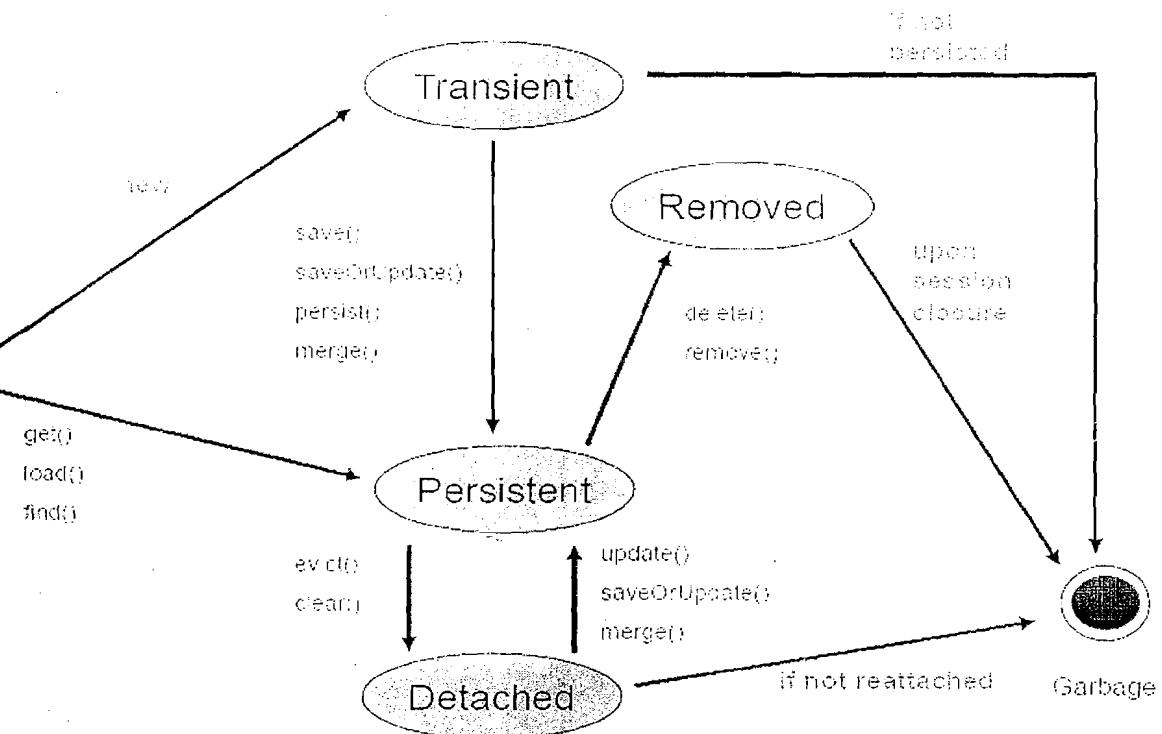
Note: We can find the different diagrams for the persistent object life cycle as follows

Hibernate-Persistent Object Lifecycle

Mr.SekharReddy



Hibernate Lifecycle



Transient state: An object is said to be in transient state, when it is not associated with session and not present in database.

Example: Table

ACCOUNT

ACONO	NAME	BALANCE
1001	kesavareddy	1500
1002	Sekhar	2000

Application code:

1. Account account = new Account();
2. account.setAccountId(1003);
3. account.setName("yellareddy");
4. account.setBalance(1500);

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

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

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

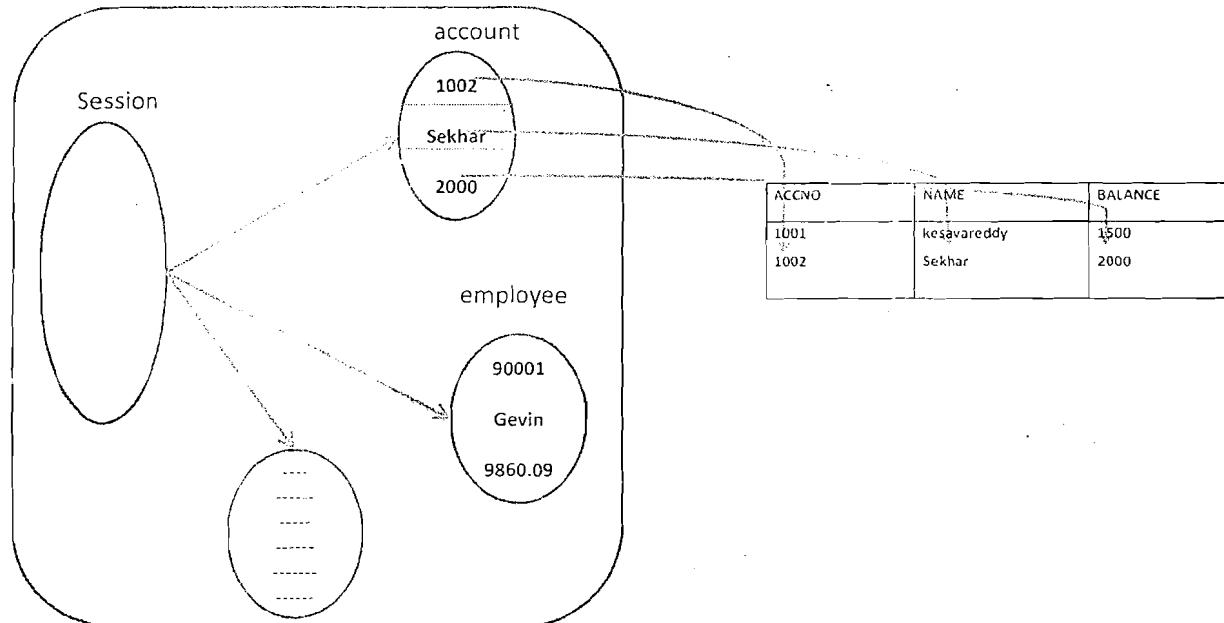
Example: Table

ACCOUNT

ACCNO	NAME	BALANCE
1001	kesavareddy	1500
1002	Sekhar	2000

Application code:

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



- In the above example **account** object is associated with session and there is a matching record in ACCOUNT table. So we can say that **account** object is in persistent state.
- In this state object is transactional. I.e. the object is synchronized with database record.
- Changes made to objects in a persistent state are automatically saved to the database without invoking session persistence methods

USE CASE: Explains Persistent state

```
Session session =SessionUtil.getSession();
session.beginTransaction();
```

// 'transient' state – Hibernate is NOT aware that it exists

```
Account account = new Account();

// transition to the 'persistent' state. Hibernate is NOW
// aware of the object and will save it to the database
session.save(account);

// modification of the object will automatically be
// saved because the object is in the 'persistent' state
account.setBalance(500);

// commit the transaction
session.getTransaction().commit();
```

USE CASE: Explains Transient state

```
Session session = SessionUtil.getSession();
session.beginTransaction();

// retrieve account with id 1. account is returned in a 'persistent' state
Account account = (Account) session.get(Account.class, 1);

// transition to the 'transient' state. Hibernate deletes the
// database record, and no longer manages the object
session.delete(account);

// modification is ignored by Hibernate since it is in the 'transient' state
account.setBalance(500);

// commit the transaction
session.getTransaction().commit();
```

```
// notice the Java object is still alive, though deleted from the database.
// stays alive until developer sets to null, or goes out of scope
System.out.println(account.getBalance());
```

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

Example: Table

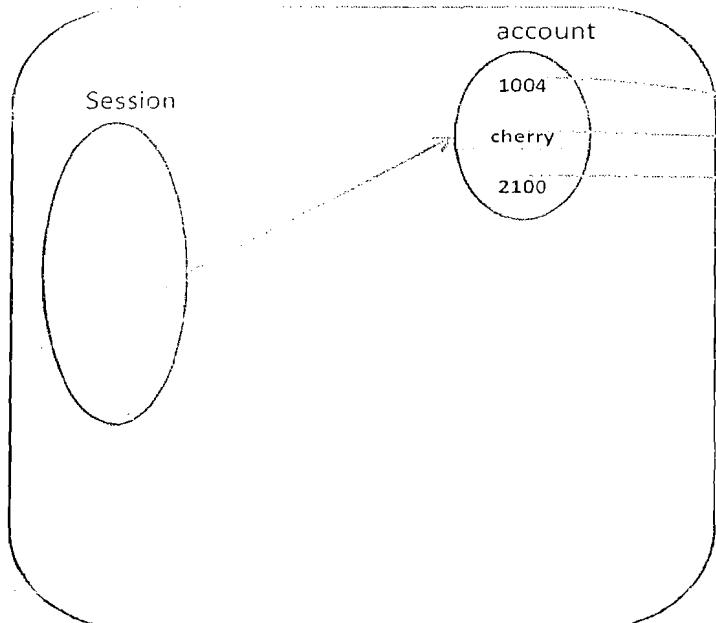
ACCOUNT

ACCNO	NAME	BALANCE
1001	kesavareddy	1500
1002	Sekhar	2000

Application code :

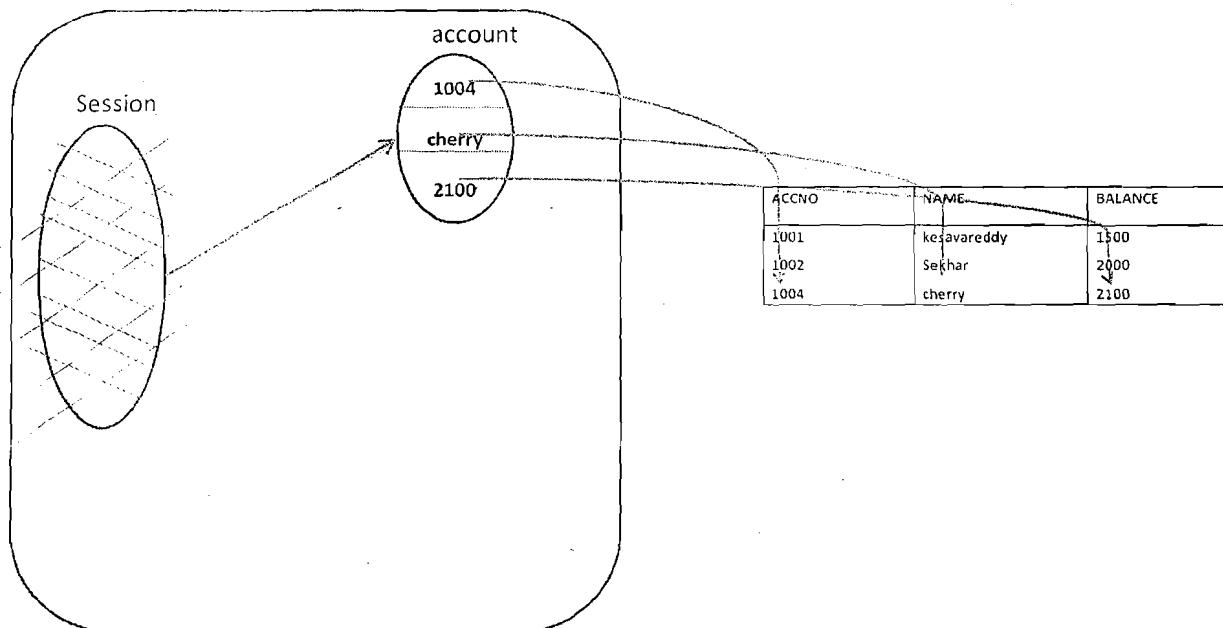
1. Account account = new Account();
2. account.setAccountId(1004);
3. account.setName("cherry");

4. account.setBalance(2100);
5. // Now the account object is said to be in transient state.
6. session.save(account);
7. // Now the account object is said to be in persistence state.



ACCNO	NAME	BALANCE
1001	kesavareddy	1500
1002	Sekhar	2000
1004	cherry	2100

8. session.close();
9. // Now the account object is said to be in detached state.



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

=> In this state object is non-transactional. Means object is not in sync with database. So Changes made to detached objects are not saved to the database.

USE CASE: Explains Detached state

```
Session session1 =SessionUtil.getSession();
session.beginTransaction();

// retrieve account with id 1. account is returned in a 'persistent' state
Account account = session.get(Account.class, 1);

// transition to the 'detached' state. Hibernate no longer manages the object
session.close();

// modification is ignored by Hibernate since it is in the
// 'detached' state, but the account still represents a row in the database
account.setBalance(500);

// commit the transaction
session.getTransaction().commit();
```

USE CASE: Explains Detached state

```
Session session1 =SessionUtil.getSession();

// retrieve account with id 1. account is returned in a 'persistent' state
Account account = session1.get(Account.class, 1);

// transition to the 'detached' state. Hibernate no longer manages the object
Session1.close();

// modification is ignored by Hibernate since it is in the
// 'detached' state, but the account still represents a row in the database
account.setBalance(500);

// re-attach the object to an open session, returning it to the
// 'persistent' state and allowing its changes to be saved to the database
Session session2 =SessionUtil.getSession();
Session2.beginTransaction();
session2.update(account);

// commit the transaction
session2.getTransaction().commit();
```

Saving Changes to the Database

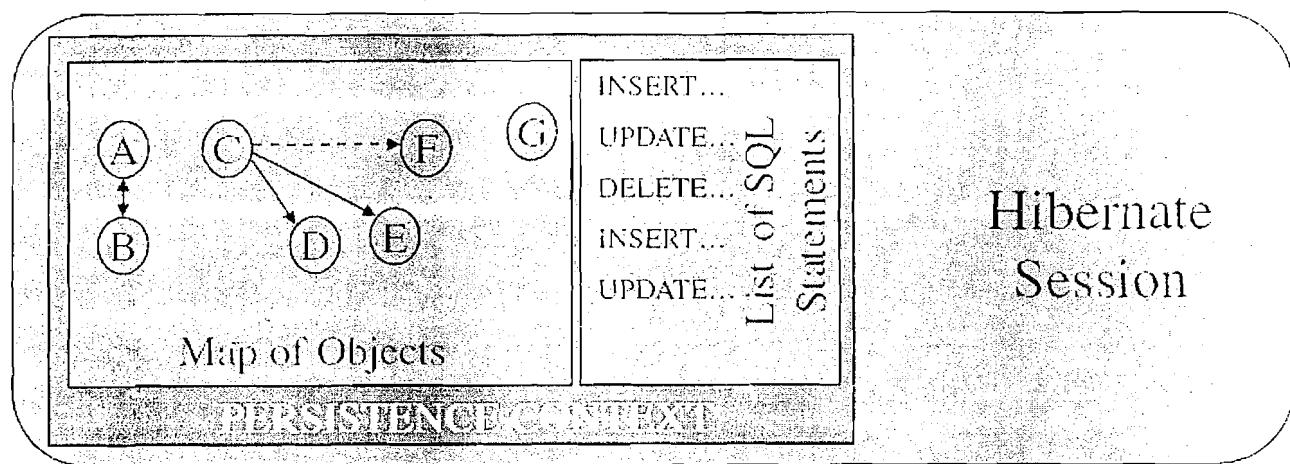
⇒ Session methods do NOT save changes to the database

- save();
- update();
- delete();

- ⇒ These methods actually **SCHEDULE** changes to be made to the database
- ⇒ Once Transaction committed, all the queries will be pushed to the database
 - session.getTransaction().commit();

The Persistence Context

- ⇒ Each Session object contains one **PersistentContext**. It might be containing the follow things:
- Graph of managed persistent instances
- List of SQL statements to send to the database



Flushing the Context

Submits the stored SQL statements to the database Occurs when:

- transaction.commit() is called
- session.flush() is called explicitly

USE CASE : Scheduled Changes

```
Session session =SessionUtil.getSession();
Session.beginTransaction();
```

```
// 'transient' state – Hibernate is NOT aware that it exists
```

```
Account account = new Account();
```

```
// Transition to the 'persistent' state. Hibernate is NOW
```

```
// aware of the object and will save it to the database
```

```
// schedules the insert statements to create the object in the database
```

```
session.save(account);
```

```
// modification of the object will automatically be saved scheduled
```

```
//because the object is in the 'persistent' state
```

```
// (actually alters the initial insert statement since it hasn't been sent yet)
```

```
account.setBalance(500);
```

```
// flushes changes to the database and commit the transaction
```

```
session.getTransaction().commit();
```

Session methods

I am going to use the following entity to explain the session methods.

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.Id;
6. import javax.persistence.Table;
7.
8. @Entity
9. @Table(name = "ACCOUNT")
10. public class Account {
11.     private int accountId;
12.     private String name;
13.     private double balance;
14.
15.     @Id
16.     @Column(name = "ACCNO")
17.     public int getAccountId() {
18.         return accountId;
19.     }
20.
21.     public void setAccountId(int accountId) {
22.         this.accountId = accountId;
23.     }
24.
25.     @Column(name = "NAME")
26.     public String getName() {
27.         return name;
28.     }
29.
30.     public void setName(String name) {
31.         this.name = name;
32.     }
33.
34.     @Column(name = "BALANCE")
35.     public double getBalance() {
36.         return balance;
37.     }
38.
39.     public void setBalance(double balance) {
40.         this.balance = balance;
41.     }
42.
43.     @Override
44.     public String toString() {
45.         return "Account [accountId=" + accountId + ", name=" + name
46.                 + ", balance=" + balance + "]";
47.     }
48.
49. }
```

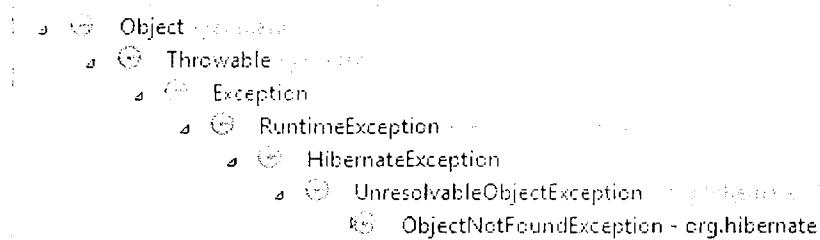
Q.) What are the differences between `load()` and `get()` methods?

Hibernate Session provides two method to access object e.g. `session.get()` and `session.load()`. Both looked quite similar to each other but there are many differences between load and get method which can affect performance of our application.

load()

1.) Throws "`org.hibernate.ObjectNotFoundException`" if object is not found in cache as well as on database.

Type hierarchy of '`org.hibernate.ObjectNotFoundException`':



As we can see the `ObjectNotFoundException` hierarchy, we can say this is un-checked exception. So we no need to write try-catch block to handle this exception.

2) It is **lazy loading**, means when we call `session.load(Class, identifier)` method it will not return entity object, it will return proxy object. When we try to access the non-identifier properties from the proxy object, at that time it will hit the database and load the entity object.

3) As `Session.load()` return proxy instance, so it is not fully available in any future detached state.

4) Use this method if it is sure that the objects exist.

5) It is just like `EntityManager.getReference()` method of JPA

get()

1) It will return '`null`' value, if object is not found on cache as well as on database.

2) It is early loading, Means when we call `session.load(Class, identifier)` method it will hit the database immediately and load entity object and return entity object.

3) As `Session.get()` returns a fully initialized instance, so it is fully available in any future detached state.

4) Use this method if it is not sure that the objects exist.

5) It is just like `EntityManager.find()` method of JPA

NOTE: If working with detached objects is not needed, `load()` or `getReference()` can be used to have better performance.

NOTE: Session.load() or EntityManager.getReference() should be used if a fully initialized instance is not needed, which saves a database roundtrip if nothing other than creation of an association is done, with the proxied instance in managed state.

NOTE: load() method exists prior to get() method which is added on user request.

Q.) How to call get and load methods ?

```
50. public void callLoad(){  
51.     Session session = SessionUtil.getSession();  
52.     session.beginTransaction();  
53.     try {  
54.         Account account = (Account) session.load(Account.class,9001);  
55.         // At this line put Break.point... Now observe the console, After this line executed,  
56.         // we can't find any select statement. And if observe on the variables window, account object not initialized.  
57.         // Now account is just a proxy object.  
58.  
59.         System.out.println(account.getName());  
60.         // After this line you can find select query on the console, And now account object is initialized with database data.  
61.  
62.     } catch (ObjectNotFoundException e) {  
63.         e.printStackTrace();  
64.     }  
65.     session.getTransaction().commit();  
66.  
67.     // System.out.println(account.getName());  
68.     // This would fail!!!  
69.  
70. }
```



```
1. public void callGet() {  
2.     Session session = SessionUtil.getSession();  
3.     session.beginTransaction();  
4.     Account account = (Account) session.get(Account.class,9001);  
5.     // At this line put Break.point... Now observe the console, After this line executed,  
6.     // we can find select statement. And if observe on the variables window,  
7.     // account object initialized with database data  
8.  
9.     System.out.println(account.getName());  
10.    session.getTransaction().commit();  
11.  
12.    // System.out.println(account.getName());  
13.    // no problem!!!  
14.  
15. }
```

When to use Session get() and load() in Hibernate

1. if object present we have to implement some logic, if not we need to implement some other logic.

get(): if the object is not there, it returns null. Then we can implement above requirement as follows

```
if(object == null){  
    //some code
```

```
 }else{  
    //some other code  
}
```

`load()`: if the object is not there it throws an exception. So we can't implement this requirement using `load()`.

For the above requirement we go for `get()` method.

2. If you want to use the JavaBean that you are retrieving from the database after the database transaction has been committed, you'll want to use the `get` method, and quite frankly, that tends to be most of the time. For example, if you load a User instance in a Servlet, and you want to pass that instance to a Java Server Page for display purposes, you'd need to use the `get` method, otherwise, you'd have a `LazyInitializationException` in your JSP.

3. `get()` method could suffer performance penalty if only identifier method like `getId()` is accessed. So consider using `load()` method if your code doesn't access any method other than identifier or you are OK with lazy initialization of object.

Overloaded load() methods (Hibernate 4.x)

- `load(Class theClass, Serializable id) : Object - Session`
- `load(Object object, Serializable id) : void - Session`
- `load(String entityName, Serializable id) : Object - Session`
- `load(Class theClass, Serializable id, LockMode lockMode) : Object`
- `load(Class theClass, Serializable id, LockOptions lockOptions) : Object`
- `load(String entityName, Serializable id, LockMode lockMode) : Object`
- `load(String entityName, Serializable id, LockOptions lockOptions) : Object`

Overloaded get() methods (Hibernate 4.x)

- `get(Class clazz, Serializable id) : Object - Session`
- `get(String entityName, Serializable id) : Object - Session`
- `get(Class clazz, Serializable id, LockMode lockMode) : Object`
- `get(Class clazz, Serializable id, LockOptions lockOptions) : Object`
- `get(String entityName, Serializable id, LockMode lockMode) : Object`
- `get(String entityName, Serializable id, LockOptions lockOptions) : Object`

Q) When update() method has to call ?

Transient state?

Transient state means the object is not associated with session and not presented in database. When there is no record in the database, no question of updating the record. So when the object is in transient state we can't call `update()` method

Persistent state?

Persistence state means the object is associated with session as well as presented in the database. If the object is in the persistent state then the object is said to be synchronized with database. So whatever modifications done to the object,

those changes will be updated in the database and vice versa. So we no need to call update() method when object is in persistent state.

Detached state?

Detached state means the object is not associated with session but presented in the database. In this state if we want to do any modifications to the object we should call update() method. Because in this state the object is not synchronized with database.

public void saveOrUpdate(Object object) throws HibernateException;

If the record is not there in the database it will try to insert the record. If the record is there in the database it will try to update the record.

Pseudo code of saveOrUpdate():

```
1. class SessionImpl implements Session{  
2.     public void saveOrUpdate(Object object){  
3.  
4.         //SELECT THE RECORD  
5.         Object obj = session.get(object);  
6.         // IF RECORD EXIST  
7.         If( obj != null){  
8.             // UPDATE THE RECORD  
9.             update(obj);  
10.        } else{ // IF RECORD NOT EXIST  
11.            // INSERT THE RECORD  
12.            save(obj);  
13.        }  
14.    }
```

Ex: Table

ACCOUNT

ACCNO	NAME	BALANCE

Application code:

```
1. Session session = SessionUtil.getSession();  
2. session.beginTransaction().begin();  
3.  
4. Account account = new Account();  
5. account.setAccountId(1001);  
6. account.setName("sekhar");  
7. account.setBalance(6800.00);  
8.  
9. session.saveOrUpdate(account);  
10.  
11. session.getTransaction().commit();  
12.  
13. //Now saveOrUpdate() internally calls save(), because record with 9001 id is not available in the database.
```

Table:

ACCOUNT

ACCNO	NAME	BAL
1001	sekhar	6800

Application code:

1. Session session = SessionUtil.getSession();
2. session.beginTransaction().begin();
- 3.
4. Account account = new Account();
5. account.setAccountId(1001);
6. account.setName("sekhar Reddy");
7. account.setBalance(8200.00);
- 8.
9. session.saveOrUpdate(account);
- 10.
11. session.getTransaction().commit();
- 12.
13. //Now saveOrUpdate() internally class update(). Because the record with 9001 id is already exists in the database.

Table:

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar Reddy	8200

public Connection close() throws HibernateException;

Once the transaction is completed we need to close the session. When we close the session all the associated objects with the session will be de-associated from session and associated JDBC connection also closed. It is not strictly necessary to close the session but you must at least **disconnect()** it.

public void clear();

This method is used to de-associate all the objects from session.

Table:

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar	8400
1002	Kesavareddy	9500

Application code:

1. Session session = SessionUtil.getSession();
2. session.beginTransaction().begin();
- 3.
4. Account account1 = (Account)session.get(Account.class,1001);
5. Account account2 = (Account)session.get(Account.class,1002);
6. // Now account1 and account2 objects are in persistent state.
- 7.
8. account1.setName("new sekhar");
9. account2.setName("new kesavareddy");
- 10.

11. session.clear();
12. // Now account1 and account2 objects are in detached state.
- 13.
14. session.getTransaction().commit();

(OR)

1. Session session = SessionUtil.getSession();
2. session.beginTransaction().begin();
- 3.
4. Account account1 = (Account)session.get(Account.class,1001);
5. Account account2 = (Account)session.get(Account.class,1002);
6. // Now account1 and account2 objects are in persistent state.
- 7.
8. session.clear();
9. // Now account1 and account2 objects are in detached state.
- 10.
11. account1.setName("new sekhar");
12. account2.setName("new kesavareddy");
- 13.
14. session.getTransaction().commit();

After execution:

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar	8400
1002	kesavareddy	9500

- In the above example, when we call session.clear() method, account1 and account2 objects will be de-associated from the session object. i.e. account1 and account2 objects are moved from persistent state to detached state.
- Now account1 and account2 are in non-transactional state. So even we are committing the transaction the modified values of account1 and account2 are not updated in the database.

public void evict(Object object) throws HibernateException:

This method is used de-associates the specified object from session.

Table:

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar	8400
1002	kesavareddy	9500

Application code:

1. Session session = SessionUtil.getSession();
2. session.beginTransaction().begin();
- 3.
4. Account account1 = (Account)session.get(Account.class,1001);
5. Account account2 = (Account)session.get(Account.class,1002);

```
6. // Now account1 and account2 objects are in persistent state.  
7.  
8. account1.setName("new sekhar");  
9. account2.setName("new kesavareddy");  
10.  
11. session.evict(account1);  
12. // Now account1 in detached state and account2 in persistent state.  
13.  
14. session.getTransaction().commit();
```

After Execution:

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar	8400
1002	new kesavareddy	9500

- evict() is used to de-associate specified object from the session object.
- In the above example when we call `session.evict(account1)` account1 object will be de-associated from session object.
- After calling `transaction.commit()`, only account2 object will be updated. Because it is in persistent state.

public boolean contains(Object object);

It is used to check whether the object is associated with session or not.

Application code:

```
1. Session session = SessionUtil.getSession();  
2.  
3. Account account = (Account)session.get(Account.class,1001);  
4. System.out.println("After calling get() method");  
5.  
6. if(session.contains(account)){  
7.     System.out.println("account is associated with session");  
8. } else{  
9.     System.out.println("account is not associated with session");  
10. }  
11.  
12. session.clear();  
13. System.out.println("After calling clear() method");  
14.  
15. if(session.contains(account)){  
16.     System.out.println("account is associated with session");  
17. } else{  
18.     System.out.println("account is not associated with session");  
19. }
```

public boolean isConnected();

To check weather there is a connection is associated with the session or not.

```
1. Session session = SessionUtil.getSession();
2. if(session.isConnected()){
3.     System.out.println("connected");
4. }else{
5.     System.out.println("not connected");
6. }
7. session.close();
8. if(session.isConnected()){
9.     System.out.println("connected");
10. }else{
11.    System.out.println("not connected");
12. }
```

public void flush() throws HibernateException;

This method is used to synchronize session data with database.

Application code:

```
1. Session session = SessionUtil.getSession();
2. session.beginTransaction().begin();
3.
4. Account account = (Account)session.get(Account.class,1001);
5. account.setName("new sekhar");
6. account.setBalance(9500);
7. session.flush();
8.
9. System.out.println("Break.. Point and observe the console..");
10.
11. session.getTransaction().commit();
```

- In the above example when we call **session.flush()**, Hibernate checks or compares account object data and corresponding record database. If it finds difference, it will execute update query to update object data into the database record.
- When **transaction.commit()** is called it will also check object data and corresponding record data. If it finds different it will update object data into database.
- So after **transaction.commit()**, we should not call **session.flush()** because when we call **transaction.commit()** session is in sync with database

Batch Processing

- The execution of series of programs is called batch processing. Batch processing is the process of reading data from a persistent store, doing something with the data, and then storing the processed data in the persistent store.
- Usually we run Batch process, when computer resources are less busy.
- We are using **flush()** and **clear()** methods of the Session API for the batch insert process.

When you need to upload a large number of records into our database by using hibernate we are using the below code.

Eg:

```
1. Session session = SessionUtil.getSession();
2. Transaction tx = session.beginTransaction();
3.
4. for ( int i=0; i<100000; i++ ) {
5.     Employee employee = new Employee(.....);
6.     session.save(employee);
7. }
8.
9. tx.commit();
10. session.close();
```

The prime step for using the batch processing feature is to set hibernate.jdbc.batch_size as batch size to a number either at 20 or 50 depending on object size. This shows the hibernate container that every X rows to be inserted as batch.

Eg:-

```
1. Session session = SessionFactory.openSession();
2. Transaction tx = session.beginTransaction();
3.
4. for ( int i=0; i<100000; i++ ) {
5.     Employee employee = new Employee(.....);
6.     session.save(employee);
7.     if( i % 50 == 0 ) { // Same as the JDBC batch size
8.         //flush a batch of inserts and release memory:
9.         session.flush();
10.        session.clear();
11.    }
12. }
13.
14. tx.commit();
15. session.close();
```

Advantage:- Batch processing helps to resolve the problem of OutOfMemoryException.

public void flush() throws HibernateException;

This method is used to synchronize the database data with session data. To understand the importance of refresh() method observe the following scenarios.

Case1: with single session, single time calling get() method:

Table:

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar	8400

Application code:

```
1. Session session = SessionUtil.getSession();
2. session.getTransaction().begin();
```

```
3.  
4. Account account = (Account) session.get(Account.class, 1001);  
5. System.out.println("Before updating the database...");  
6. System.out.println("Name : " + account.getName());  
7. System.out.println("Balance : " + account.getBalance());  
8.  
9. // Break..point... go to database and modify the data  
10.
```

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar new	9500

```
11. System.out.println("After updating the database ...");  
12. System.out.println("Name : " + account.getName());  
13. System.out.println("Balance : " + account.getBalance());  
14.  
15. session.getTransaction().commit();  
16.  
17. session.close();
```

Output:

Before updating the database...

Name : sekhar

Balance : 8400.0

After updating the database...

Name : sekhar

Balance : 8400.0

Explanation:

- When we call the **get()** on **session** object, it will hit the database and get the data from the database and creates entity object and assign the retrieved data to entity object. And finally that entity object will be cached on the session object.
- When we update the data on the database it will not get the updated data. Just it always shows session cached data.

Case2: with single session, multiple times calling get() method:

Table:

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar	8400

Application code:

```
1. Session session = SessionUtil.getSession();  
2. session.beginTransaction();  
3.  
4. Account account = (Account) session.get(Account.class, 1001);  
5. System.out.println("Before updating the database...");
```

```
6. System.out.println("Name : " + account.getName());
7. System.out.println("Balance : " + account.getBalance());
8.
9. // Break..point... go to database and modify the data
10.
```

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar new	9500

```
11. account = (Account) session.get(Account.class, 1001);
12. System.out.println("After updating the database ... ");
13. System.out.println("Name : " + account.getName());
14. System.out.println("Balance : " + account.getBalance());
15.
16. session.getTransaction().commit();
17.
18. session.close();
```

Output:

Before updating the database...

Name : sekhar
Balance : 8400.0
After updating the database...

Name : sekhar
Balance : 8400.0

Explanation:

- When we call the **get()** on **session** object(second time), it will check whether the object is available in **session** or not. If the object is available in **session**, it will not hit the database.
- In above example with Accno 1001 already **account** object is already available in **session** object. That's why even we call **get()** method on **session** object 2nd time, it will not hit the database. That's why it didn't display the updated record data of database, instead it displayed previous data only.

Case 3: creating multiple sessions.

Table:

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar	8400

Application code:

```
1. Session session1 = SessionUtil.getSession();
2. Session session2 = SessionUtil.getSession();
3.
4. Account account = (Account) session1.get(Account.class, 1001);
5. System.out.println("Before updating the database... ");
6. System.out.println("Name : " + account.getName());
7. System.out.println("Balance : " + account.getBalance());
```

- 8.
9. // Break..point... go to database and modify the data

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar new	9500

10. account = (Account) session2.get(Account.class, 1001);
11. System.out.println("After updating the database ...");
12. System.out.println("Name : " + account.getName());
13. System.out.println("Balance : " + account.getBalance());

Output:

Before updating the database...

Name : sekhar

Balance : 8400.0

After updating the database...

Name : sekhar new

Balance : 9500.0

Explanation:

- In the above example, **session2** object doesn't have any associated objects. That's why when we call **get()** method on **session2**, it hit the database and executes the select query and retrieve the record and display the updated record of database.
- But here every time we are creating new **session** object to get the updated Record. To solve the above problem we can use **refresh()** method.

Case4: using refresh()

Table:

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar	8400

Application code:

1. Session session = SessionUtil.getSession();
- 2.
3. Account account = (Account) session.get(Account.class, 1001);
4. System.out.println("Before updating the database...");
5. System.out.println("Name : " + account.getName());
6. System.out.println("Balance : " + account.getBalance());
- 7.
8. // Break..point... go to database and modify the data

ACCOUNT

ACCNO	NAME	BALANCE

1001	sekhar new	9500
------	------------	------

```
9. session.refresh(account);
10. System.out.println("After updating the database ...");
11. System.out.println("Name : " + account.getName());
12. System.out.println("Balance : " + account.getBalance());
```

Output:

Before updating the database...

Name : sekhar

Balance : 8400.0

After updating the database...

Name : sekhar new

Balance : 9500.0

Explanation:

- In the above example when we call **refresh()**, Hibernate compares database data and object data. If it finds any difference it will again execute select query and update the object data.

public Object merge(Object object) throws HibernateException;

Consider the following example,

Table:

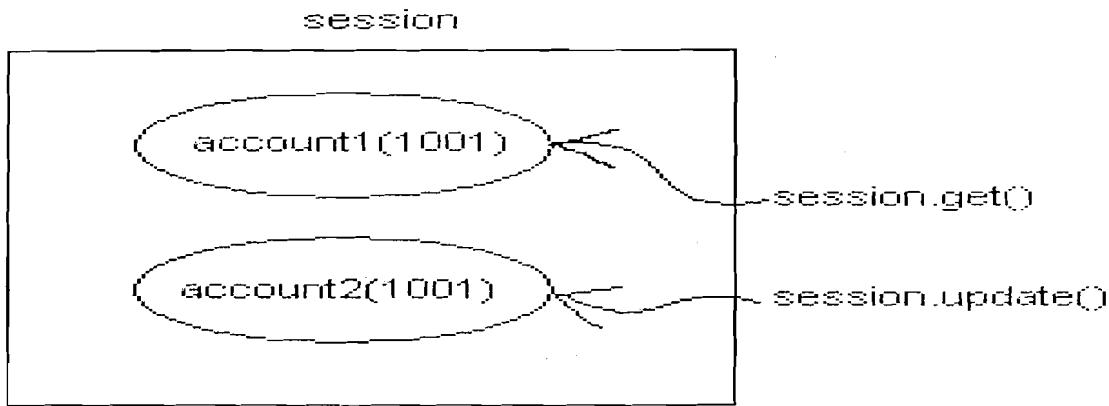
ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar	8400

Application Code:

```
1. session.beginTransaction().begin();
2.
3. Account account1=(Account)session.get(Account.class, 1001);
4.
5. Account account2= new Account();
6. account2.setAccountId(1001);
7. account2.setName("cherry");
8. account2.setBalance(6500);
9.
10. session.update(account2);
11.
12. session.getTransaction().commit();
```

Output: org.hibernate.NonUniqueObjectException: a different object with the same identifier value was already associated with the session: [com.sekharit.hibernate.entity.Account#1001]



- We can't place two different objects(of same type) with the same identifier in the session object.
- In the above example, by calling get() method **account1** object with identifier '1001' will be there in session. And by calling update() method **account2** object with identifier '1001' is also trying to come into session object. It is the problem. To avoid this we will go for merge().
- In the above example, If we use merge() method instead of update() method, we won't get exception. Just **account2** object data will be updated into database.
- merge() method behave differently in different scenarios. merge() method can insert, update, merge the data. To understand more clear about merge() method consider the following cases.

Case 1: merge() method insert the data

Table:

ACCOUNT

ACCNO	NAME	BALANCE
1001	sekhar	8400

Application Code:

1. Session session = SessionUtil.getSession();
2. session.beginTransaction();
- 3.
4. Account account = new Account();
5. account.setAccountId(1002);
6. account.setName("cherry");
7. account.setBalance(4500.00);
- 8.
9. session.merge(account);
- 10.
11. session.getTransaction().commit();
- 12.
13. session.close();

After execution:

ACCOUNT

ACCNO	NAME	BALANCE
1001	Sekhar	8400
1002	cherry	4500

- In the above example, when we call merge() method, first it will try to load Account object with identifier 1002, As we don't have a record in ACCOUNT table with ACCNO#1002, it will insert Account(1002, cherry, 4500.00) object into database.

Case 2: merge() method update the data

Table:

ACCOUNT

ACCNO	NAME	BALANCE
1001	Sekhar	8400
1002	cherry	4500

Application Code:

1. Session session = SessionUtil.getSession();
2. session.beginTransaction().begin();
- 3.
4. Account account = new Account();
5. account.setAccountId(1002);
6. account.setName("yellareddy");
7. account.setBalance(5600.00);
- 8.
9. session.merge(account);
- 10.
11. session.getTransaction().commit();
- 12.
13. session.close();

After execution:

ACCOUNT

ACCNO	NAME	BALANCE
1001	Sekhar	8400
1002	yellareddy	5600

- In the above example, when we call merge() method, first it will try to load Account object with identifier 1002, As we have a record in ACCOUNT table with ACCNO#1002, it will try to update with latest Account(1002, yellareddy, 5600.00) object into database.

Case 3: merge() method merge the detached object data into persistent object

Table:

ACCOUNT

ACCNO	NAME	BALANCE
1001	Sekhar	8400
1002	yellareddy	5600

Application Code:

1. Session session = SessionUtil.getSession();
2. session.beginTransaction().begin();
- 3.
4. Account account1=(Account)session.get(Account.class, 1001);

```

5.
6. Account account2= new Account();
7. account2.setAccountId(1001);
8. account2.setName("kesavareddy");
9. account2.setBalance(6500);
10.
11. session.merge(account2);
12.
13. session.getTransaction().commit();
14. session.close();

```

After execution:

ACCOUNT

ACCNO	NAME	BALANCE
1001	kesavareddy	6500
1002	yellareddy	5600

- In the above example, Before 11th line, **account1** is in **Persistent** state, and **account2** in **detached** state.
- In the above example, when we call **merge()** method it will check, weather there is any object associated with the session with same identifier(1001).
- In our example, **account1#1001** object is already associated with session, So merge() method now, Copy the state of **account2#1001** object state into **account1#1001** object. After 11th line also, **account1** is in **Persistent** state, and **account2** in **detached** state.
- When the transaction is committed, As account1#1001(Persistent-state) data is modified, so it will hit the update query, to update session data with database.

Q) What is the difference between merge and update?

update () : When the session does not contain an persistent instance with the same identifier, and if it is sure use update for the data persistence in hibernate.

merge (): Irrespective of the state of a session, if there is a need to save the modifications at any given time, use **merge()**.

public Serializable getIdentifier(Object object) throws HibernateException;

To know the object identifier value at the runtime, we need to call **getIdentifier(Object object)**

Application code:

```

Account account = (Account)session.get(Account.class,1001);
Serializable id = session.getIdentifier (account);
System.out.println("Identifier of Account is :" + id);

```

Transaction methods:

We can apply Transaction management by using transaction object.

We can get the Transaction object in 2 ways.

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

```
Transaction transaction = session.getTransaction();
```

beginTransaction() return different transaction objects. For each and every request transaction object creates and begins new transaction context.

getTransaction() return same transaction object for every request. We need to call **begin()** on transaction object to begin the transaction.

public void persist(Object object) throws HibernateException;

- This method is same as **save()**, but **save(object)** returns identifier and **persist(object)** doesn't return any value.
- When we are using generator classes to generate the identifier, At that time if we want to know identifier value which is generated by generator class, then we can go for **save()** method.
- When we don't want to know generated identifier, then we can for **persist()** method.

Method Signatures:

```
Session
  -> save(Object) : Serializable
  -> save(String, Object) : Serializable
```

```
Session
  -> persist(Object) : void
  -> persist(String, Object) : void
```

public void replicate(Object object, ReplicationMode replicationMode) throws HibernateException;

This method is used to move the object from detached state to persistent state.

ReplicationMode has attributes described below.

- **ReplicationMode.OVERWRITE** : this mode reads the processing request and affect the result in database. After committing the transaction the results are stored in database.
- **ReplicationMode.IGNORE** : this mode ignores the processed request. Means it doesn't affect the result in database.
- **ReplicationMode.LATEST_VERSION** : this mode reads the processing request and affect the result in database.
- **ReplicationMode.EXCEPTION** : this mode reads the processing request and affect the result the database.

Application Code:

1. Session session = SessionUtil.getSession();
2. session.getTransaction().begin();
- 3.

```
4.     Account account = (Account) session.get(Account.class, 1001);
5.     System.out.println("After get() method");
6.     if (session.contains(account)) {
7.         System.out.println("associated");
8.     } else {
9.         System.out.println("not-associated");
10.    }
11.
12.    session.evict(account);
13.    System.out.println("After evict() method");
14.    if (session.contains(account)) {
15.        System.out.println("associated");
16.    } else {
17.        System.out.println("not-associated");
18.    }
19.
20.    session.replicate(account, ReplicationMode.LATEST_VERSION);
21.    System.out.println("After replicate() method");
22.    if (session.contains(account)) {
23.        System.out.println("associated");
24.    } else {
25.        System.out.println("not-associated");
26.    }
27.
28.    session.getTransaction().commit();
29.
30.    session.close();
```

public void lock(Object object, LockMode lockMode) throws HibernateException;

If we are updating some group of tables no other person is allowed to update the records on the same table until our work is completed. So using lock(), if we lock the record it doesn't allow updating from different users till we commit the transaction.

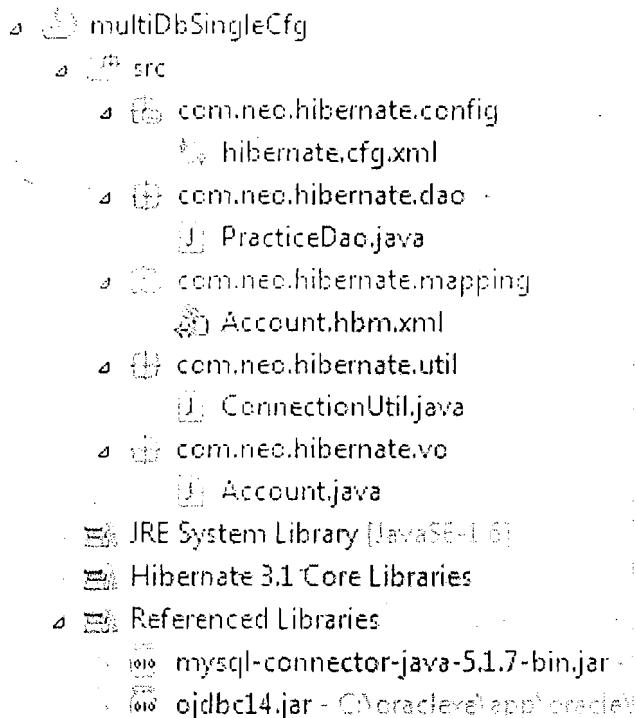
- Hibernate defines Several lock modes
 - LockMode.NONE : Don't go to the database unless the object isn't in either cache.
 - LockMode.READ : Bypass both levels of cache, and perform a version check to verify the object in memory is the same version as that currently exists in the database.
 - LockMode.UPGRADE: Bypass both levels of cache, and perform a version check and obtain a database-level pessimistic lock upgrade lock, if that is supported.
 - LockMode.UPGRADE_NOWAIT: same as UPGRADE , but use a SELECT FOR UPDATE NOWAIT on Oracle. This disables waiting for concurrent lock releases, thus throwing a locking exception can't be obtained.
 - LockMode.WRITE: Is obtained automatically when Hibernate has written to a row in the current Transaction. This is an internal mode and can't be specified explicitly.

NOTE: this method got deprecated in Hibernate 4.x version

Q.) How to work with MYSQL database?

- ⇒ install the MYSQL database.
- ⇒ Select All Programs → MYSQL → MYSQL 5.0 → mysql command line client
- ⇒ Give password
- ⇒ In the MYSQL prompt give the following commands
 - Create database mydb
 - Use mydb
- ⇒ Now perform database operations normally.
- ⇒ When you open mysql command line client again just give **password**, and give **use mydb** command. (don't give **create database mydb**)

Q.) How to connect to multiple Databases using Single Configuration File in Hibernate?



Oracle DataBase Table:

SQL> desc account;

Name	Null?	Type
ACNO		NUMBER(5)
NAME		VARCHAR2(10)
BAL		NUMBER(8,2)

MySQL DataBase Table

Field	Type	Null	Key	Default	Extra
acno	int(5)	YES		NULL	
name	char(15)	YES		NULL	
bal	float	YES		NULL	

Account.java

```

1. package com.neo.hibernate.vo;
2.
3. public class Account {
4.     private int accno;
5.     private String name;
6.     private double balance;
7.
8.     public int getAccno() {
9.         return accno;
10.    }
11.
12.    public void setAccno(int accno) {
13.        this.accno = accno;
14.    }
15.
16.    public String getName() {
17.        return name;
18.    }
19.
20.    public void setName(String name) {
21.        this.name = name;
22.    }
23.
24.    public double getBalance() {
25.        return balance;
26.    }
27.
28.    public void setBalance(double balance) {
29.        this.balance = balance;
30.    }
31.
32. }
```

PracticeDao.java

```
1. package com.neo.hibernate.dao;
2.
3. import org.hibernate.Session;
4. import org.hibernate.SessionFactory;
5. import org.hibernate.cfg.Configuration;
6.
7. import com.neo.hibernate.util.ConnectionUtil;
8. import com.neo.hibernate.vo.Account;
9.
10. public class PracticeDao {
11.     public static void main(String[] args) {
12.
13.         Configuration configuration = new Configuration();
14.         configuration.configure("com/neo/hibernate/config/hibernate.xml");
15.         SessionFactory factory = configuration.buildSessionFactory();
16.         // Oracle
17.         Session oracleSession =
18.             factory.openSession(ConnectionUtil.getOracleConnection());
19.         Account oracleAccount = (Account)
20.             oracleSession.get(Account.class, 1001);
21.         System.out.println("Oracle Account table details ....");
22.         System.out.println("Accno : "+oracleAccount.getAccno());
23.         System.out.println("Name : "+oracleAccount.getName());
24.         System.out.println("Balance : "+oracleAccount.getBalance());
25.         //Mysql
26.         Session mysqlSession =
27.             factory.openSession(ConnectionUtil.getMysqlConnection());
28.         Account mysqlAccount= (Account)mysqlSession.get(Account.class, 1001);
29.         System.out.println("Mysql Account table details ....");
30.         System.out.println("Accno : "+mysqlAccount.getAccno());
31.         System.out.println("Name : "+mysqlAccount.getName());
32.         System.out.println("Balance : "+mysqlAccount.getBalance());
33.
34.     }
35. }
36.
```

ConnectionUtil.java

```
1. package com.neo.hibernate.util;
2.
3. import java.sql.Connection;
4. import java.sql.DriverManager;
5. import java.sql.SQLException;
6.
7. public class ConnectionUtil {
8.     static {
9.         try {
10.             Class.forName("oracle.jdbc.driver.OracleDriver");
11.             Class.forName("com.mysql.jdbc.Driver");
12.         } catch (Exception e) {
13.             e.printStackTrace();
14.         }
15.     }
16. }
```

```
16.     }
17.
18.     public static Connection getOracleConnection() {
19.         Connection connection = null;
20.         try {
21.             connection = DriverManager.getConnection(
22.                 "jdbc:oracle:thin:@localhost:1521:XE", "system", "tiger");
23.         } catch (SQLException e) {
24.             e.printStackTrace();
25.         }
26.         return connection;
27.     }
28.
29.     public static Connection getMysqlConnection() {
30.         Connection connection = null;
31.         try {
32.             connection = DriverManager.getConnection(
33.                 "jdbc:mysql://localhost:3306/mydb", "root", "tiger");
34.         } catch (SQLException e) {
35.             e.printStackTrace();
36.         }
37.         return connection;
38.     }
39.
40. }
```

Account.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
3.
4. <hibernate-mapping>
5.   <class name="com.neo.hibernate.vo.Account" table="ACCOUNT">
6.     <id name="accno" column="ACNO"></id>
7.     <property name="name"></property>
8.     <property name="balance" column="BAL"></property>
9.   </class>
10. </hibernate-mapping>
```

hibernate.cfg.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD
   3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
3.
4. <hibernate-configuration>
5.   <session-factory>
6.     <property name="dialect">org.hibernate.dialect.OracleDialect</property>
7.     <mapping resource="com/neo/hibernate/mapping/Account.hbm.xml" />
8.   </session-factory>
9. </hibernate-configuration>
```

Q.)How to connect to multiple Databases using Multiple Configuration Files in Hibernate?

```
o .\multiDBMultiCfg
  o .\src
    o .\com.nec.hibernate.config
      o .\mysql.hibernate.cfg.xml
      o .\oracle.hibernate.cfg.xml
    o .\com.nec.hibernate.dao
      o .\PracticeDao.java
    o .\com.nec.hibernate.mapping
      o .\Account.hbm.xml
    o .\com.nec.hibernate.util
      o .\SessionUtil.java
    o .\com.nec.hibernate.vo
      o .\Account.java
  o .\JRE System Library...
  o .\Hibernate 3.1 Core Libraries
  o .\Referenced Libraries...
    o .\ojdbc14.jar
    o .\mysql-connector-java-5.1.7-bin.jar
    ...
```

Oracle DataBase Table:

SQL> desc account;

Name	Null?	Type
ACNO		NUMBER(5)
NAME		VARCHAR2(10)
BAL		NUMBER(8,2)

MySQL DataBase Table:

Field	Type	Null	Key	Default	Extra
acno	int(5)	YES		NULL	
name	char(15)	YES		NULL	
bal	float	YES		NULL	

Account.java

```
1. package com.neo.hibernate.vo;
2.
3. public class Account {
4.     private int accno;
5.     private String name;
6.     private double balance;
7.
8.     public int getAccno() {
9.         return accno;
10.    }
11.
12.    public void setAccno(int accno) {
13.        this.accno = accno;
14.    }
15.
16.    public String getName() {
17.        return name;
18.    }
19.
20.    public void setName(String name) {
21.        this.name = name;
22.    }
23.
24.    public double getBalance() {
25.        return balance;
26.    }
27.
28.    public void setBalance(double balance) {
29.        this.balance = balance;
30.    }
31.
32. }
```

PracticeDao.java

```
1. package com.neo.hibernate.dao;
2. import org.hibernate.Session;
3. import com.neo.hibernate.util.SessionUtil;
4. import com.neo.hibernate.vo.Account;
5.
6. public class PracticeDao {
7.     public static void main(String[] args) {
8.         // Oracle
9.         Session oracleSession = SessionUtil.getOracleSession();
10.        Account oracleAccount = (Account)oracleSession.get(Account.class,
11.            1001);
12.        System.out.println("Oracle Account table details ....");
13.        System.out.println("Accno : "+oracleAccount.getAccno());
14.        System.out.println("Name : "+oracleAccount.getName());
15.        System.out.println("Balance : "+oracleAccount.getBalance());
16.        //Mysql
17.        Session mysqlSession = SessionUtil.getMysqlSession();
18.        Account mysqlAccount= (Account)mysqlSession.get(Account.class, 1001);
19.        System.out.println("Mysql Account table details ....");
20.        System.out.println("Accno : "+mysqlAccount.getAccno());
```

```
21. System.out.println("Name : "+mysqlAccount.getName());
22. System.out.println("Balance : "+mysqlAccount.getBalance());
23. }
24. }
```

SessionUtil.java

```
1. package com.neo.hibernate.util;
2.
3. import org.hibernate.Session;
4. import org.hibernate.SessionFactory;
5. import org.hibernate.cfg.Configuration;
6. import org.jaxen.function.ConcatFunction;
7.
8. public class SessionUtil {
9.
10.     private static SessionFactory oraclFactory;
11.     private static SessionFactory mysqlFactory;
12.     static {
13.         oraclFactory = new Configuration().configure(
14.             "com/neo/hibernate/config/oracle.hibernate.cfg.xml")
15.             .buildSessionFactory();
16.         mysqlFactory = new Configuration().configure(
17.             "com/neo/hibernate/config/mysql.hibernate.cfg.xml")
18.             .buildSessionFactory();
19.     }
20.
21.     public static Session getOracleSession() {
22.         return oraclFactory.openSession();
23.     }
24.
25.     public static Session getMysqlSession() {
26.         return mysqlFactory.openSession();
27.     }
28.     public static void main(String[] args) {
29.         System.out.println(getMysqlSession());
30.     }
31. }
```

Account.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
3.
4. <hibernate-mapping>
5.   <class name="com.neo.hibernate.vo.Account" table="ACCOUNT">
6.     <id name="accno" column="ACNO"></id>
7.     <property name="name"></property>
8.     <property name="balance" column="BAL"></property>
9.   </class>
10. </hibernate-mapping>
```

mysql.hibernate.cfg.xml

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

```
2. <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD  
3.0//EN"  
3. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  
4.  
5.   <hibernate-configuration>  
6.     <session-factory>  
7.       <property name="dialect">org.hibernate.dialect.MySQLDialect</property>  
8.       <property name="connection.url">jdbc:mysql://localhost:3306/mydb</property>  
9.       <property name="connection.username">root</property>  
10.      <property name="connection.password">tiger</property>  
11.      <property name="connection.driver_class">com.mysql.jdbc.Driver</property>  
12.      <mapping resource="com/neo/hibernate/mapping/Account.hbm.xml" />  
13.    </session-factory>  
14.  </hibernate-configuration>
```

oracle.hibernate.cfg.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>  
2. <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD  
3.0//EN"  
3. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  
4.   <hibernate-configuration>  
5.     <session-factory>  
6.       <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>  
7.       <property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>  
8.       <property name="connection.username">system</property>  
9.       <property name="connection.password">tiger</property>  
10.      <property  
11.        name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>  
12.      <mapping resource="com/neo/hibernate/mapping/Account.hbm.xml" />  
13.    </session-factory>  
14.  </hibernate-configuration>
```

Generator classes

- Generator classes are used to generate the 'identifier' for a persistent object. i.e. While saving an object into the database, the generator informs to the hibernate that, how the primary key value for the new record is going to generate
- Hibernate using different primary key generator algorithms, for each algorithm internally a java class is there for its implementation
- All Generator classes has to implement 'org.hibernate.id.IdentifierGenerator" interface, And has to override `generate()` method. The logic to generate 'identifier' has to write in this method.
- In built-in generator classes, identifier generation logic has implemented by using JDBC.
- If we want we can write the user defined generator class and it should implement 'org.hibernate.id.IdentifierGenerator' interface and has to override `generate()` method.
- To configure generator class we can use `<generator />` tag, which is the sub element of `<id/>` tag
- `<generator />` tag has one attribute called "class" with which we can specify generator class-name.
- While configuring `<generator />` tag in mapping file, if we need to pass any parameters to generator class then we can use `<param />` tag, which is the sub element of `<generator />` tag.

Example: HBM

```
<hibernate-mapping>
  <class ... >
    <id .... >
      <generator class="generator-class-name">
        <param name="param-name">param-value </param>
      </generator>
    </id>
  </class>
</hibernate-mapping>
```

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

1. sequence
2. assigned
3. increment
4. hilo
5. seqhilo

6. identify
7. native
8. uuid
9. guid
10. select
11. foregin

In the above generators list, are used for int,long,short types of primary keys, and uuid, guid are used when the primary key column type is String type (varchar2)

1.) sequence (org.hibernate.id.SequenceGenerator)

- This generator class is database dependent it means, we cannot use this generator class for all the database, we should know whether the database supports sequence or not before we are working with it
- Not has the support with MySql
- Here we write a sequence and it should be configured in HBM file and while persisting the object in the database sequence is going to generate the identifier and it will assign to the Id property of persistent object, Then it will store the persistent object into Database.

NOTE: MYSQL even won't allow to create SEQUENCE Object in that database. Then where is the question of calling that from hibernate application?

NOTE: When we configured generator class for an entity, then we no need to assign identifier value to entity object while saving the entity. Even we assign the identifier value to entity, it will not consider our assigned value, it will use generator class generated value as identifier value. It applies to all generator classes.

Steps to work with "sequence" generator

Step 1: Create a sequence

```
SQL> CREATE SEQUENCE ACCNO_SEQ START WITH 1000 INCREMENT BY 1
```

To get next value

```
SQL> SELECT ACCNO_SEQ.NEXTVAL FROM DUAL;
```

To get current value

```
SQL> SELECT ACCNO_SEQ.CURRVAL FROM DUAL;
```

Step 2: configure the sequence in hbm file.

Account.hbm.xml

```
<hibernate-mapping>
<class .....>
    <id name="accountId" column="ACNO">
        <generator class="sequence" >
            <paramname="sequence">ACCNO_SEQ</param>
        </generator>
    </id>
    <property ...../>
</class>
</hibernate-mapping>
```

(OR)

```
<id name="accountId " column="ACNO">
    <generator class="org.hibernate.id.SequenceGenerator" >
        <param name="sequence">ACCNO_SEQ </param>
    </generator>
</id>
```

(OR)

```
<id name="accountId " column="ACCNO">
    <generator class="sequence" />
</id>
```

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

NOTE: But remember, if we enable **hbm2ddl.auto** property in hibernate configuration file, then hibernate will create the database objects if they are not exist.

NOTE: It is not advisable to use the default sequence, always prefer to create a different sequence to each entity separately.

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

```
1. Session session = SessionUtil.getSession();
2. session.beginTransaction();
3.
4. Account account = new Account();
5. account.setName("sekhar");
6. account.setBalance(5000);
7.
8. Serializable id = session.save(account);
9.
10. System.out.println("Account is created with accno : "+id);
11. session.getTransaction().commit();
```

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

Internal Code

```
public class SequenceGenerator implements PersistentIdentifierGenerator, Configurable {
    public static final String SEQUENCE = "sequence";
    .....
    .....

    public void configure(..){
        sequenceName = PropertiesHelper.getString(SEQUENCE, params,
            "hibernate_sequence"); // default sequence name
        parameters = params.getProperty(PARAMETERS);
    }

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

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

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

- This generator supports in all the databases
- This is the default generator class used by the hibernate, if we do not specify <generator/> element under <id/> element, then hibernate by default assumes it as "assigned" generator class.
- If generator class is assigned, then the programmer is responsible for assigning the identifier value to entity before saving into the database

HBM:

```
<id name="accountId" column="ACCNO">
    <generator class="assigned" />
</id>
(OR)

<id name="accountId" column="ACCNO">
    <generator class="org.hibernate.id.Assigned" />
</id>
(OR)
```

```
<id name=" accountId " column="ACCNO" />
```

Internal Code:

```
public class Assigned implements IdentifierGenerator, Configurable {  
    public Serializable generate(...){  
        final Serializable id = ... get the developer given id.  
        if (id==null) {  
            throw new IdentifierGenerationException(  
                "ids for this class must be manually assigned before calling save(): " + entityName);  
        }  
        return id;  
    }  
}
```

NOTE: While testing don't assign identifier value to entity and then try to save entity then it throws **IdentifierGenerationException**.

NOTE: While testing with assigned generator class, identifier type in entity should be some object type rather than primitive type.

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

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

HBM:

```
<id name=" accountId " column="ACCNO">  
    <generator class="increment" />  
</id>  
    (OR)  
<id name=" accountId " column="ACCNO">  
    <generator class=" org.hibernate.id.IncrementGenerator " />  
</id>
```

Internal Code:

```
public class IncrementGenerator implements IdentifierGenerator, Configurable {  
  
    public synchronized Serializable generate(...){  
  
        if (sql!=null) {  
            getNext( session );  
        }  
        return ... final number after adding '1'  
    }  
}
```

```

    }

    public void configure( . . . ) {
        . . .
        . . .

        sql = "select max(" + column + ") from " + buf.toString();
    }

    private void getNext( SessionImplementor session ) {

        PreparedStatement st = ....prepareSelectStatement(sql);
        ResultSet rs = st.executeQuery();
        if ( rs.next() ) {
            next = rs.getLong(1) + 1;
        }
    }
}

```

The HiLo Algorithm

The HiLo (High/Low) algorithm knows how to generate unique number series using two values: the high and the low. The high value is used as a base for a series (or range) of numbers, while the size of this series is donated by the low value. A unique series is generated using the following steps:

- 1) Load the and atomically increment the high value
- 2) Multiple the high value by the low value ($\text{high} * \text{low}$), the result is the first number (lower bound) of the current series
- 3) The last number (higher bound) of the current series is donated by the following calculation: $(\text{high} * \text{low}) + \text{low} - 1$
- 4) When a client needs to obtain a number the next one from the current is used, once the entire series has been exhausted the algorithm goes back to step 1

Example: suppose that the current high value in the database is 52 and the low value is configured to be 32,767. When the algorithm starts it loads the high value from the database and increments it in the same transaction (the new high value in the database is now 53). The range of the current numbers series can now be calculated:

- Lower bound = $52 * 32767 = 1,703,884$
- Upper bounds = $1,703,884 + 32,767 - 1 = 1,736,650$

All of the numbers in the range of 1,703,884 to 1,736,650 can be safely allocated to clients, once this keys pool has been exhausted the algorithm needs to access the database again to allocate a new keys pool. This time the high value is 53 (immediately incremented to 54) and the keys range is:

- Lower bound = $53 * 32,767 = 1,736,651$
- Upper bounds = $1,736,651 + 32,767 - 1 = 1,769,417$

And so on

The big advantage of this algorithm is keys preallocation which can dramatically improve performance. Based on the low

value we can control the database hit ratio. As illustrated using the 32,767 we hit the database only once in a 32,767 generated keys. The downside (at least by some people - but in my opinion this is a none-issue) is that each time the algorithm restarts it leaves a 'hole' in the keys sequence.

Hibernate has several HiLo based generators: TableHiLoGenerator, MultipleHiLoPerTableGenerator, SequenceHiLoGenerator

TableHiLoGenerator

A simple HiLo generator, uses a table to store the HiLo high value. The generator accepts the following parameters

- **table** - the table name, defaults to 'hibernate_unique_key'
- **column** - the name of the column to store the next high value, defaults to 'next_hi'
- **max_low** - the low number (the range) defaults to 32,767 (Short.MAX_VALUE)

MultipleHiLoPerTableGenerator

A table HiLo generator which can store multiple key sets (multiple high values each for a different entity). This is useful when we need each entity (or some of the entities) has its own keys range. It supports the following parameters:

- **table** - the table name, default to 'hibernate_sequences'
- **primary_key_column** - key column name, defaults to 'sequence_name'
- **value_column** - the name of the column to store the next high value, defaults to 'sequence_next_hi_value'
- **primary_key_value** - key value for the current entity (or current keys set), default to the entity's primary table name
- **primary_key_length** - length of the key column in DB represented as a varchar, defaults to 255
- **max_low** - the low numer (the range) defaults to 32,767 (Short.MAX_VALUE)

The generator uses a single table to store multiple high values (multiple series), when having multiple entities using the same generator Hibernate matches an entity to a high value using the primary_key_value which is usually the entity name. A sample table can look like

sequence_name	sequence_next_high_value
ENTITY1	234
ENTITY2	876
ENTITY3	8

SequenceHiLoGenerator

A simple HiLo generator but instead of a table uses a sequence as the high value provider.

- **sequence** - the sequence name, defaults to 'hibernate_sequence'
- **max_low** - the low number (the range) defaults to 9.

4.) hilo(org.hibernate.id.TableHiLoGenerator)

- This generator is database independent
- hilo uses a hi/lo algorithm to generate identifiers.
- HiLo algorithm generate identifiers based on the given table and column(stores high value). Default table is 'hibernate_unique_key' column is 'next_hi'.

HBM:

```
<id name="accountId" column="ACCNO">
    <generator class="hilo" >
        <param name="table">HIGH_VAL_TAB</param>
        <param name="column">HIGH_VAL_COL</param>
        <param name="max_lo">60</param>
    </generator>
</id>
(OR)
<id name="accountId" column="ACCNO">
    <generator class=" org.hibernate.id.TableHiLoGenerator" >
        <param name="table">HIGH_VAL_TAB</param>
        <param name="column">HIGH_VAL_COL</param>
        <param name="max_lo">60</param>
    </generator>
</id>
```

Internal code

```
class TableHiLoGenerator extends TableGenerator {
    .....
}

class TableGenerator {
    public static final String COLUMN = "column";
    public static final String DEFAULT_COLUMN_NAME = "next_hi";
    public static final String TABLE = "table";
    public static final String DEFAULT_TABLE_NAME = "hibernate_unique_key";
    .....
}
```

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

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

HBM:

```
<id name="accountId" column="ACNO">
    <generator class="seqhilo" >
        <param name="sequence">ACCNO_SEQ</param>
        <param name="max_lo">5</param>
    </generator>
</id>
(OR)

<id name=" accountId " column="ACCNO">
    <generator class="seqhilo" >
        <param name="org.hibernate.id.SequenceHiLoGenerator ">ACCNO_SEQ</param>
        <param name="max_lo">5</param>
    </generator>
</id>
```

Q.) How to work with MYSQL database?

- ⇒ install the MYSQL database.
- ⇒ Select All Programs → MYSQL → MYSQL 5.0 → mysql command line client
- ⇒ Give password
- ⇒ In the MYSQL prompt give the following commands
 - create database mydb
 - use mydb
- ⇒ Now perform database operations normally.
- ⇒ When we login to database next time onwards don't give create database command, just give use command.

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

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

Syntax to create identity columns in MYSQL database:

```
CREATE TABLE STUDENT(
SNO INT(10) NOT NULL AUTO_INCREMENT,
COURSE CHAR(20),
FEE FLOAT,
NAME CHAR(20),
PRIMARY KEY (SNO)
)
```

HBM:

```
<id name=" accountId " column="ACNO">
    <generator class="identity" />
</id>
```

Example: To create identity columns in MYSQL database.

```

C:\Program Files\MySQL\MySQL Server 5.0\bin\mysql.exe
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.0.51b-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database myschema;
Query OK, 1 row affected (0.00 sec)

mysql> use myschema;
Database changed
mysql> CREATE TABLE STUDENT(SNO INT<10> NOT NULL AUTO_INCREMENT, NAME CHAR<20>, COURSE CHAR<15>);
ERROR 1075 (42000): Incorrect table definition; there can be only one auto column and it must be defined as a key
mysql> CREATE TABLE STUDENT(SNO INT<10> NOT NULL AUTO_INCREMENT, NAME CHAR<20>, COURSE CHAR<15>, PRIMARY KEY(SNO));
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO STUDENT(NAME, COURSE) VALUES('sekhar', 'spring');
Query OK, 1 row affected (0.05 sec)

mysql> select * from student;
+----+----+----+
| SNO | NAME | COURSE |
+----+----+----+
| 1   | sekhar | spring |
+----+----+----+
1 row in set (0.00 sec)

mysql> INSERT INTO STUDENT(NAME, COURSE) VALUES('sekhar', 'spring');
Query OK, 1 row affected (0.03 sec)

mysql> select * from student;
+----+----+----+
| SNO | NAME | COURSE |
+----+----+----+
| 1   | sekhar | spring |
| 2   | sekhar | spring |
+----+----+----+
2 rows in set (0.00 sec)

mysql> INSERT INTO STUDENT(NAME, COURSE) VALUES('sekhar', 'spring');
Query OK, 1 row affected (0.02 sec)

mysql> select * from student;
+----+----+----+
| SNO | NAME | COURSE |
+----+----+----+
| 1   | sekhar | spring |
| 2   | sekhar | spring |
| 3   | sekhar | spring |
+----+----+----+
3 rows in set (0.00 sec)

mysql>

```

7.) native

- native is not having any generator class because, it uses internally identity or sequence or hilo generator classes.
- native picks up identity or sequence or hilo generator class depending upon the capabilities of the underlying database.

HBM:

```

<id name="accountId" column="ACNO">
  <generator class="native" >
    <param name="sequence">ACCNO_SEQ</param>
    <param name="table">HIGH_VAL_TAB</param>
    <param name="column">HIGH_VAL_COL</param>

```

```
<param name="max_lo">60</param>
</generator>
</id>
```

NOTE: test with mysql and oracle

- ⇒ If we connect to mysql it takes identity.
- ⇒ If we connect to oracle it takes sequence.

8.) uuid(org.hibernate.id.UUIDHexGenerator)

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

HBM:

```
<id name="accountId" column="ACNO">
    <generator class="uuid" >
    </generator>
</id>
```

(OR)

```
<id name=" accountId " column="ACCNO">
    <generator class="org.hibernate.id.UUIDHexGenerator " >
    </generator>
</id>
```

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

guid: uses a database generated guid string on MS-SQL and MYSQL.

foreign: foreign uses the identifier of another associated object. Usually uses in conjunction with a <one-to-one> primary key association.

User Defined Generator class

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

Steps to implement user defined generator class.

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

Step2: Give the generator class entry in the <generator> tag of HBM file.

Step 1: AccountNumberGenerator.java

```
1. package com.sekharit.hibernate.id;
2.
3. import java.io.Serializable;
4. import java.sql.Connection;
5. import java.sql.ResultSet;
6. import java.sql.Statement;
7.
8. import org.hibernate.HibernateException;
9. import org.hibernate.engine.SessionImplementor;
10. import org.hibernate.id.IdentifierGenerator;
11.
12. public class AccountNumberGenerator implements IdentifierGenerator {
13.
14.     public Serializable generate(SessionImplementor session, Object object)
15.             throws HibernateException {
16.         String accno = "SBH";
17.         try {
18.             Connection connection = session.connection();
19.             String query = "SELECT TO_CHAR("
20.                         + "ACCNO_SEQ.NEXTVAL,'FM00000000') FROM DUAL";
21.             Statement statement = connection.createStatement();
22.             ResultSet resultSet = statement.executeQuery(query);
23.             if (resultSet.next()) {
24.                 accno = accno + resultSet.getString(1);
25.             }
26.
27.         } catch (Exception e) {
28.             e.printStackTrace();
29.         }
30.         return accno;
31.     }
32. }
```

Step 2:HBM

```
<id name="accountId" column="ACCNO">
    <generator class="com.sekharit.hibernate.id.AccountNumberGenerator" />
</id>
```

NOTE: In the above user defined generate class I used database sequence to generate next value... And I got the connection from session object, but this method is deprecated in latest version. So we need to physically create the connection in the latest versions.

JPA supported Identifier Generators

The @Id annotation lets you define which property is the identifier of your entity bean. This property can be set by the application itself or be generated by Hibernate (preferred). You can define the identifier generation strategy using **@GeneratedValue** annotation:

- **AUTO** - either **identity** column, **sequence** or **table** depending on the underlying Database
- **TABLE** - hilo algorithm
- **IDENTITY** - identity column
- **SEQUENCE** - sequence

NOTE: Hibernate provides more id generators than the basic JPA.

1) Without any Generator:

Student.java

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @Column(name = "SNO")
7.     private Integer sid;
8.     @Column(name = "NAME")
9.     private String name;
10.    @Column(name = "COURSE", length = 20)
11.    private String course;
12.    @Column(name = "FEE")
13.    private double fee;
14.
15.    // getters & setters
16. }
```

StudentDAO.java

```

1. public class StudentDAO {
2.     public static void main(String[] args) {
3.         Session session = SessionUtil.getSession();
4.         session.beginTransaction();
5.
6.         for(int i = 1 ; i <= 5 ;i++){
7.             Student student = new Student();
8.             student.setCourse("java-"+i);
9.             student.setName("sekhar-"+i);
10.            student.setFee(111*i);
11.            session.save(student);
12.        }
13.
14.        session.getTransaction().commit();
15. }
```

```
16.      }
17. }
```

ERROR:

org.hibernate.id.IdentifierGenerationException: ids for this class must be manually assigned before calling save():

NOTE : So if we don't specify any Id generation strategy, By default hibernate internally takes it as assigned generator.

2) With @GeneratedValue annotation

- ⇒ In this approach **@GeneratedValue** annotation default value is "**strategy=GenerationType.AUTO**"

```
1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GeneratedValue
7.     @Column(name = "SNO")
8.     private Integer sid;
9.     @Column(name = "NAME")
10.    private String name;
11.    @Column(name = "COURSE", length = 20)
12.    private String course;
13.    @Column(name = "FEE")
14.    private double fee;
15.
16.    // getters & setters
17. }
```

NOTE: As we didn't sepcify any strategy, by default it will take **strategy=GenerationType.AUTO**. As we know **AUTO** work like "**native**" generator, so it will take either **identity** or **sequence** or **hilo**. As we are working with Oracle Database, it will take **sequence** generator. As we didn't specify any sequence name, by default it will take sequence name as "**hibernate_sequence**" So, we can write the above entity as follows.

```
1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GeneratedValue(strategy=GenerationType.AUTO)
7.     @Column(name = "SNO")
8.     private Integer sid;
9.     @Column(name = "NAME")
```

```

10.     private String name;
11.     @Column(name = "COURSE", length = 20)
12.     private String course;
13.     @Column(name = "FEE")
14.     private double fee;
15.
16.     // getters & setters
17. }
```

If we want our own seqncence, then we need to configure sequence name also....

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @SequenceGenerator(name="myGenerator", sequenceName="STUDENT_SEQ")
7.     @GeneratedValue(strategy=GenerationType.AUTO, generator="myGenerator")
8.     @Column(name = "SNO")
9.     private Integer sid;
10.    @Column(name = "NAME")
11.    private String name;
12.    @Column(name = "COURSE", length = 20)
13.    private String course;
14.    @Column(name = "FEE")
15.    private double fee;
16.
17.    // getters & setters
18. }
```

- ⇒ This strategy supports either **identity column**, **sequence** or **table hilo generator** depending on the underlying DB
- ⇒ If we work with Oracle database, It will consider sequence, because Oracle database doesn't supports **identity columns**
- ⇒ If we work with MYSQL database, It will consider **identity**, because MYSQL database supports **identity columns**

Syntax to create identity columns in MYSQL database :

```

CREATE TABLE STUDENT(
SNO INT(10) NOT NULL AUTO_INCREMENT,
COURSE CHAR(20),
FEE FLOAT,
NAME CHAR(20),
PRIMARY KEY (SNO)
)
```

NOTE: It is similar to "**native**" generator of hibernate

2) With @GeneratedValue(strategy=GenerationType.SEQUENCE)

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @SequenceGenerator(name="myGenerator", sequenceName="STUDENT_SEQ")
7.     @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="myGenerator")
8.     @Column(name = "SNO")
9.     private Integer sid;
10.    @Column(name = "NAME")
11.    private String name;
12.    @Column(name = "COURSE", length = 20)
13.    private String course;
14.    @Column(name = "FEE")
15.    private double fee;
16.
17.    // getters & setters
18. }
```

- ⇒ This is supported by Oracle, It will take the given sequence and by executing the sequence it will retrieve the identifier.
- ⇒ This is not supported by MYSQL

3) With @GeneratedValue(strategy=GenerationType.IDENTITY)

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GeneratedValue(strategy=GenerationType.IDENTITY)
7.     @Column(name = "SNO")
8.     private Integer sid;
9.     @Column(name = "NAME")
10.    private String name;
11.    @Column(name = "COURSE", length = 20)
12.    private String course;
13.    @Column(name = "FEE")
14.    private double fee;
15.
16.    // getters & setters
17. }
```

- ⇒ This is not supported by Oracle, But supported by MYSQL database

4) With @GeneratedValue(strategy=GenerationType.TABLE)

```
1. @Entity
```

```

2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @TableGenerator(
7.         name="myGenerator",
8.         initialValue=2000,
9.         allocationSize=1,
10.        table="PK_VALUE_TAB",
11.        pkColumnName="PK_COLUMN",
12.        pkColumnValue="PK_VALUE",
13.        valueColumnName="PK_VALUE_COLUMN")
14.     @GeneratedValue(strategy=GenerationType.TABLE, generator="myGenerator")
15.     @Column(name = "SNO")
16.     private Integer sid;
17.     @Column(name = "NAME")
18.     private String name;
19.     @Column(name = "COURSE", length = 20)
20.     private String course;
21.     @Column(name = "FEE")
22.     private double fee;
23.
24.     // getters & setters
25. }

```

Hibernate supported Identifier Generators

To integrate hibernate specific generators into JPA we can use either **@org.hibernate.annotations.GenericGenerator** or **@org.hibernate.annotations.GenericGenerators** annotations.

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

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GenericGenerator(name="myGenerator", strategy="assigned")
7.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
8.     @Column(name = "SNO")
9.     private Integer sid;
10.    @Column(name = "NAME")
11.    private String name;
12.    @Column(name = "COURSE", length = 20)
13.    private String course;
14.    @Column(name = "FEE")
15.    private double fee;
16.
17.    // getters & setters
18. }

```

OR

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GenericGenerator(name="myGenerator", strategy="org.hibernate.id.Assigned ")
7.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
8.     @Column(name = "SNO")
9.     private Integer sid;
10.    @Column(name = "NAME")
11.    private String name;
12.    @Column(name = "COURSE", length = 20)
13.    private String course;
14.    @Column(name = "FEE")
15.    private double fee;
16.
17.    // getters & setters
18. }
```

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

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GenericGenerator(name="myGenerator", strategy="increment ")
7.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
8.     @Column(name = "SNO")
9.     private Integer sid;
10.    @Column(name = "NAME")
11.    private String name;
12.    @Column(name = "COURSE", length = 20)
13.    private String course;
14.    @Column(name = "FEE")
15.    private double fee;
16.
17.    // getters & setters
18. }
```

3.)sequence (org.hibernate.id.SequenceGenerator)

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GenericGenerator(name="myGenerator", strategy="sequence ")
```

```

7. @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
8. @Column(name = "SNO")
9. private Integer sid;
10. @Column(name = "NAME")
11. private String name;
12. @Column(name = "COURSE", length = 20)
13. private String course;
14. @Column(name = "FEE")
15. private double fee;
16.
17. // getters & setters
18. }

```

NOTE: As there are no parameters it will take default sequence name(HIBERNATE_SEQUENCE)

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GenericGenerator(
7.         name="myGenerator",
8.         strategy="sequence"
9.         parameters={@Parameter(name="sequence",value="STUDENT_SEQ")})
10.    )
11.    @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
12.    @Column(name = "SNO")
13.    private Integer sid;
14.    @Column(name = "NAME")
15.    private String name;
16.    @Column(name = "COURSE", length = 20)
17.    private String course;
18.    @Column(name = "FEE")
19.    private double fee;
20.
21.    // getters & setters
22. }

```

4.) User defined generator:(com.sekharit.hibernate.id.AccnoGenerator)

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GenericGenerator(
7.         name="myGenerator",
8.         strategy=" com.sekharit.hibernate.id.StudnetNumberGenerator "
9.     )
10.    @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
11.    @Column(name = "SNO")
12.    private Integer sid;

```

```

13. @Column(name = "NAME")
14. private String name;
15. @Column(name = "COURSE", length = 20)
16. private String course;
17. @Column(name = "FEE")
18. private double fee;
19.
20. // getters & setters
21. }

```

5.) hilo(org.hibernate.id.TableHiLoGenerator)

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GenericGenerator(
7.         name="myGenerator",
8.         strategy="sequence"
9.         parameters={
10.             @Parameter(name="table", value="PK_VALUE_TAB"),
11.             @Parameter(name="column", value="PK_VALUE_COLUMN")
12.         }
13.     )
14.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
15.     @Column(name = "SNO")
16.     private Integer sid;
17.     @Column(name = "NAME")
18.     private String name;
19.     @Column(name = "COURSE", length = 20)
20.     private String course;
21.     @Column(name = "FEE")
22.     private double fee;
23.
24.     // getters & setters
25. }

```

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

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GenericGenerator(name="myGenerator", strategy="identity ")
7.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
8.     @Column(name = "SNO")
9.     private Integer sid;
10.    @Column(name = "NAME")

```

```

11. private String name;
12. @Column(name = "COURSE", length = 20)
13. private String course;
14. @Column(name = "FEE")
15. private double fee;
16.
17. // getters & setters
18. }

```

7.) native(identity/sequence/hilo)

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GenericGenerator(
7.         name="myGenerator",
8.         strategy="native"
9.         parameters={
10.             @Parameter(name="sequence",value="STUDENT_SEQ"),
11.             @Parameter(name="table", value="PK_VALUE_TAB"),
12.             @Parameter(name="column", value="PK_VALUE_COLUMN")
13.         }
14.     )
15.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
16.     @Column(name = "SNO")
17.     private Integer sid;
18.     @Column(name = "NAME")
19.     private String name;
20.     @Column(name = "COURSE", length = 20)
21.     private String course;
22.     @Column(name = "FEE")
23.     private double fee;
24.
25.     // getters & setters
26. }

```

8.) uuid(org.hibernate.id.UUIDHexGenerator)

```

1. @Entity
2. @Table(name = "STUDENT")
3. public class Student {
4.
5.     @Id
6.     @GenericGenerator(name="myGenerator", strategy="uuid ")
7.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
8.     @Column(name = "SNO")
9.     private String sid;
10.    @Column(name = "NAME")
11.    private String name;
12.    @Column(name = "COURSE", length = 20)

```

```
13. private String course;
14. @Column(name = "FEE")
15. private double fee;
16.
17. // getters & setters
18. }
```

NOTE: @GenericGenerator and @GenericGenerators can be used in package level annotations, making them application level generators (just like if they were in a JPA XML file).

```
@GenericGenerators(
{
    @GenericGenerator(
        name="hibseq",
        strategy = "seqhilo",
        parameters = {
            @Parameter(name="max_lo", value = "5"),
            @Parameter(name="sequence", value="heybabhey")
        }
    ),
    @GenericGenerator(...)
}
)
package org.hibernate.test.model
```

Composite ID

⇒ Process of defining primary key on more than one column is called composite id.

⇒ Usage is combination of values should not be repeated.

⇒ Means

Id1 id2

1001 2001

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

Means combination should not be repeated.

⇒ Generally this concept implemented in link tables (many-to-many relationship)

⇒ We can implement this composite id in two ways.

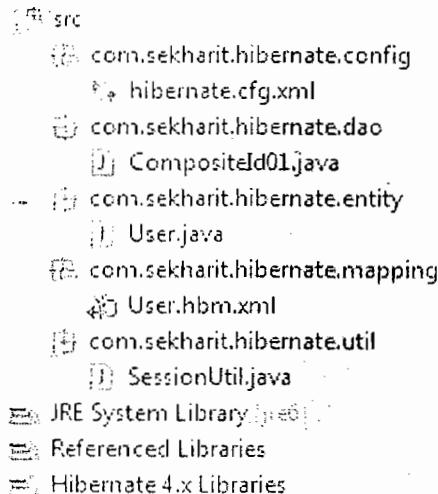
○ 1st way

- Writing primary key column related properties and other properties in the same entity.
- Example : CompositeId01(Refer example below)

○ 2nd way

- Writing primary key column related properties in one entity class and other properties in other entity class.
- Example : CompositeId02(Refer example below)

CompositeId01



SYSTEM.USER_TAB		
USER_ID	NUMBER (10)	Ø IDX_1
USER_NAME	VARCHAR2 (255)	Ø IDX_1
PASSWORD	VARCHAR2 (255)	
PROFESSION	VARCHAR2 (255)	
CITY	VARCHAR2 (255)	

User.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import java.io.Serializable;
4.
5. import javax.persistence.Column;
6. import javax.persistence.Entity;
7. import javax.persistence.Id;
8. import javax.persistence.Table;
9.
10. @Entity
11. @Table(name = "USER_TAB")
12. public class User implements Serializable {
13.     @Id
14.     @Column(name = "USER_ID")
15.     private Integer userId;
16.     @Id
17.     @Column(name = "USER_NAME")
18.     private String username;
19.     @Column(name = "PASSWORD")
20.     private String password;
21.     @Column(name = "PROFESSION")
22.     private String profession;
23.     @Column(name = "CITY")
24.     private String city;
25.
26.     // getters & setters
27.
28. }
```

User.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.     <class name="com.sekharit.hibernate.entity.User" table="USER_TAB">
6.         <composite-id>
7.             <key-property name="userId" column="USER_ID"></key-property>
8.             <key-property name="username" column="USER_NAME"></key-property>
9.         </composite-id>
10.        <property name="password" column="PASSWORD"></property>
11.        <property name="profession" column="PROFESSION"></property>
12.        <property name="city" column="CITY"></property>
13.    </class>
14. </hibernate-mapping>
```

Compositoid01.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import java.io.Serializable;
4.
5. import org.hibernate.Session;
6. import org.hibernate.Transaction;
7.
```

```
8. import com.sekharit.hibernate.entity.User;
9. import com.sekharit.hibernate.util.SessionUtil;
10.
11. public class CompositeId01 {
12.
13.     public static void main(String[] args) {
14.         Session session = SessionUtil.getSession();
15.         Transaction transaction = session.beginTransaction();
16.         // Inserting
17.
18.         User user = new User();
19.         user.setUserId(1001);
20.         user.setUsername("sekharit@gmil.com");
21.         user.setPassword("*****");
22.         user.setProfession("Senior Engineer");
23.         user.setCity("Gunipalli");
24.         Serializable id = session.save(user);
25.         System.out.println("Successfully inserted with id :" + id);
26.
27.         // Loading
28.
29.         // User user = new User();
30.         // user.setUserId(1001);
31.         // user.setUsername("sekharit@gmil.com");
32.         // user = (User) session.get(User.class, user);
33.         // System.out.println("UserId : " + user.getUserId());
34.         // System.out.println("Username : " + user.getUsername());
35.         // System.out.println("Password : " + user.getPassword());
36.         // System.out.println("Profession : " + user.getProfession());
37.         // System.out.println("City : " + user.getCity());
38.
39.         // Updating
40.
41.         // User user = new User();
42.         // user.setUserId(1001);
43.         // user.setUsername("sekharit@gmil.com");
44.         // user = (User) session.get(User.class, user);
45.         // user.setCity("Anantapur");
46.         // user.setProfession("Doctor");
47.
48.         // Deleting
49.
50.         // User user = new User();
51.         // user.setUserId(1001);
52.         // user.setUsername("sekharit@gmil.com");
53.         // user = (User) session.get(User.class, user);
54.         // session.delete(user);
55.
56.         transaction.commit();
57.
58.     }
59.
60. }
```

```
↳ .\src
  ↳ com.sekharit.hibernate.cfg
    ↳ hibernate.cfg.xml
  ↳ com.sekharit.hibernate.dao
    ↳ CompositeId02.java
  ↳ com.sekharit.hibernate.entity
    ↳ User.java
    ↳ UserPK.java
  ↳ com.sekharit.hibernate.mapping
    ↳ User.hbm.xml
  ↳ com.sekharit.hibernate.util
    ↳ SessionUtil.java
  └ JRE System Library [icc]
  └ Referenced Libraries
  └ Hibernate 4.x Libraries
```

CREATE TABLE	
USER_ID	NUMBER(10) ① IDX_1
USER_NAME	VARCHAR2(255) ① IDX_1
PASSWORD	VARCHAR2(255)
PROFESSION	VARCHAR2(255)
CITY	VARCHAR2(255)

User.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import java.io.Serializable;
4.
5. import javax.persistence.AttributeOverride;
6. import javax.persistence.AttributeOverrides;
7. import javax.persistence.Column;
8. import javax.persistence.EmbeddedId;
9. import javax.persistence.Entity;
10. import javax.persistence.Table;
11. @Entity
12. @Table(name="USER_TAB")
13. public class User implements Serializable {
14.
15.     @EmbeddedId
16.     @AttributeOverrides({
17.         @AttributeOverride(name="userId", column=@Column(name="USER_NO")),
18.         @AttributeOverride(name="username", column=@Column(name="USER_NM"))
19.     })
20.     private UserPK userPK;
21.     @Column(name="PASSWORD")
22.     private String password;
23.     @Column(name="PROFESSION")
24.     private String profession;
25.     @Column(name="CITY")
```

```
26.     private String city;
27.
28. // getters & setters
29.
30. }
```

UserPK.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import java.io.Serializable;
4.
5. import javax.persistence.Column;
6. import javax.persistence.Embeddable;
7.
8. @Embeddable
9. public class UserPK implements Serializable {
10.     @Column(name="USER_ID")
11.     private Integer userId;
12.     @Column(name="USER_NAME")
13.     private String username;
14.
15. // getters & setters
16.
17. }
```

User.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
3. <hibernate-mapping>
4.
5.   <class name="com.sekharit.hibernate.entity.User" table="USER_TAB">
6.
7.     <composite-id name="userPK" class="com.sekharit.hibernate.entity.UserPK">
8.       <key-property name="userId" column="USER_ID"></key-property>
9.       <key-property name="username" column="USER_NAME"></key-property>
10.      </composite-id>
11.
12.      <property name="password" column="PASSWORD"></property>
13.      <property name="profession" column="PROFESSION"></property>
14.      <property name="city" column="CITY"></property>
15.    </class>
16.  </hibernate-mapping>
```

Compositeld02.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import java.io.Serializable;
4.
5. import org.hibernate.Session;
6. import org.hibernate.Transaction;
7.
8. import com.sekharit.hibernate.entity.User;
9. import com.sekharit.hibernate.entity.UserPK;
10. import com.sekharit.hibernate.util.SessionUtil;
```

```
11.
12.  public class CompositeId02 {
13.
14.      public static void main(String[] args) {
15.          Session session = SessionUtil.getSession();
16.          Transaction transaction = session.beginTransaction();
17.          // Inserting
18.
19.          UserPK userPK = new UserPK();
20.          userPK.setUserId(1001);
21.          userPK.setUsername("sekharit@gmil.com");
22.
23.          User user = new User();
24.          user.setUserPK(userPK);
25.          user.setPassword("*****");
26.          user.setProfession("Senior Engineer");
27.          user.setCity("Gunipalli");
28.          Serializable id = session.save(user);
29.          System.out.println("Successfully inserted with id :" + id);
30.
31.          // Loading
32.          // UserPK userPK = new UserPK();
33.          // userPK.setUserId(1001);
34.          // userPK.setUsername("sekharit@gmil.com");
35.          // User user = (User) session.get(User.class, userPK);
36.          // System.out.println("UserId : " + user.getUserPK().getUserId());
37.          // System.out.println("Username : " + user.getUserPK().getUsername());
38.          // System.out.println("Password : " + user.getPassword());
39.          // System.out.println("Profession : " + user.getProfession());
40.          // System.out.println("City : " + user.getCity());
41.
42.          // Updating
43.
44.          // UserPK userPK = new UserPK();
45.          // userPK.setUserId(1001);
46.          // userPK.setUsername("sekharit@gmil.com");
47.          // User user = (User) session.get(User.class, userPK);
48.          // user.setCity("Anantapur");
49.          // user.setProfession("Doctor");
50.
51.          // Deleting
52.
53.          // UserPK userPK = new UserPK();
54.          // userPK.setUserId(1001);
55.          // userPK.setUsername("sekharit@gmil.com");
56.          // User user = (User) session.get(User.class, userPK);
57.          // session.delete(user);
58.
59.          transaction.commit();
60.
61.      }
62.
63.  }
```

Q) Save Student, Employee information into the following tables.

Database tables :

STUDENT

SNO	SNAME	COURSEFEE	HOUSE_NUMBER	STREET	CITY
-----	-------	-----------	--------------	--------	------

EMPLOYEE

ENO	ENAME	SALARY	DESIGNATION	HOUSE_NUMBER	STREET	CITY
-----	-------	--------	-------------	--------------	--------	------

JAVA objects:

```
public class Student {  
    private int sno;  
    private String sname;  
    private String course;  
    private double fee;  
    private String houseNumber;  
    private String street;  
    private String city;  
    // getters & setters  
}
```

```
public class Employee {  
    private int eno;  
    private String ename;  
    private double salary;  
    private String designation;  
    private String houseNumber;  
    private String street;  
    private String city;  
    // getters & setters  
}
```

- ⇒ But in the both objects houseNumber, street, city are repeated. So keep them in one pojo called Address.
- ⇒ And make that Address object as dependency to both objects.

```
public class Student {  
    private int sno;  
    private String sname;  
    private String course;  
    private double fee;  
    private Address address;  
    // getters & setters  
}
```

```
public class Employee {  
    private int eno;  
    private String ename;  
    private double salary;
```

```

private String designation;
private Address address;
// getters & setters
}

public class Address {
    private String houseNumber;
    private String street;
    private String city;
    // getters & setters
}

```

```

d component_mismatch
d src
d com.sekharit.hibernate.config
d hibernate.cfg.xml
d com.sekharit.hibernate.dao
d CreateDAO1.java
d CreateDAO2.java
d DeleteDAO1.java
d DeleteDAO2.java
d RetrieveDAO1.java
d RetrieveDAO2.java
d UpdateDAO1.java
d UpdateDAO2.java
d com.sekharit.hibernate.entity
d Address.java
d Employee.java
d Student.java
d com.sekharit.hibernate.mapping
d Employee.hbm.xml
d Student.hbm.xml
d com.sekharit.hibernate.util
d SessionUtil.java
d JRE System Library [JRE]
d Hibernate 4.x Libraries
d Referenced Libraries

```

EMP	
ENO	NUMBER(10)
NAME	VARCHAR2(1020)
DESIG	VARCHAR2(1020)
SAL	FLOAT(126)
HOUSE_NUMBER	VARCHAR2(1020)
STREET	VARCHAR2(1020)
CITY	VARCHAR2(1020)

STUDENT	
SNO	NUMBER(10)
NAME	VARCHAR2(1020)
COURSE	VARCHAR2(1020)
FEE	FLOAT(126)
HOUSE_NUMBER	VARCHAR2(1020)
STREET	VARCHAR2(1020)
CITY	VARCHAR2(1020)

Student.java

1. package com.sekharit.hibernate.entity;
- 2.
3. import javax.persistence.AttributeOverride;
4. import javax.persistence.AttributeOverrides;
5. import javax.persistence.Column;
6. import javax.persistence.Embedded;
7. import javax.persistence.Entity;
8. import javax.persistence.GeneratedValue;
9. import javax.persistence.GenerationType;
10. import javax.persistence.Id;
11. import javax.persistence.Table;

```
12.
13. import org.hibernate.annotations.GenericGenerator;
14.
15. @Entity
16. @Table(name = "STUDENT")
17. public class Student {
18.
19.     @Id
20.     @GenericGenerator(name = "myGenerator", strategy = "increment")
21.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
22.     @Column(name = "SNO")
23.     private int sno;
24.     @Column(name = "NAME")
25.     private String name;
26.     @Column(name = "COURSE")
27.     private String course;
28.     @Column(name = "FEE")
29.     private double fee;
30.     @Embedded
31.     @AttributeOverrides( {
32.         @AttributeOverride(name = "houseNumber", column = @Column(name = "S_HOUSE_NO")),
33.         @AttributeOverride(name = "street", column = @Column(name = "S_STREET")),
34.         @AttributeOverride(name = "city", column = @Column(name = "S_CITY")) })
35.     private Address address;
36.
37.
38.     //getters & setters
39. }
```

Employee.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.AttributeOverride;
4. import javax.persistence.AttributeOverrides;
5. import javax.persistence.Column;
6. import javax.persistence.Embedded;
7. import javax.persistence.Entity;
8. import javax.persistence.GeneratedValue;
9. import javax.persistence.GenerationType;
10. import javax.persistence.Id;
11. import javax.persistence.Table;
12.
13. import org.hibernate.annotations.GenericGenerator;
14.
15. @Entity
16. @Table(name = "EMP")
17. public class Employee {
18.     @Id
19.     @GenericGenerator(name = "myGenerator", strategy = "increment")
20.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
21.     @Column(name = "ENO")
22.     private int eno;
23.     @Column(name = "NAME")
24.     private String name;
25.     @Column(name = "SAL")
```

```
26.     private double salary;
27.     @Column(name = "DESIG")
28.     private String designation;
29.     @Embedded
30.     @AttributeOverrides( {
31.         @AttributeOverride(name = "houseNumber", column = @Column(name = "E_HOUSE_NO")),
32.         @AttributeOverride(name = "street", column = @Column(name = "E_STREET")),
33.         @AttributeOverride(name = "city", column = @Column(name = "E_CITY")) })
34.     private Address address;
35.
36.
37. //getters & setters
38. }
```

Address.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Embeddable;
5.
6. @Embeddable
7. public class Address {
8.     @Column(name = "HOUSE_NO")
9.     private String houseNumber;
10.    @Column(name = "STREET")
11.    private String street;
12.    @Column(name = "CITY")
13.    private String city;
14.
15. //getters & setters
16. }
```

Employee.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
3. <hibernate-mapping>
4.   <class name="com.sekharit.hibernate.entity.Employee" table="EMP">
5.
6.     <id name="eno" column="ENO">
7.       <generator class="increment"></generator>
8.     </id>
9.     <property name="name" column="NAME"></property>
10.    <property name="designation" column="DESIG"></property>
11.    <property name="salary" column="SAL"></property>
12.
13.    <component name="address" class="com.sekharit.hibernate.entity.Address">
14.      <property name="houseNumber" column="HOUSE_NUMBER"></property>
15.      <property name="street" column="STREET"></property>
16.      <property name="city" column="CITY"></property>
17.    </component>
18.
19.  </class>
20. </hibernate-mapping>
```

Student.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
3. <hibernate-mapping>
4.   <class name="com.sekharit.hibernate.entity.Student" table="STUDENT">
5.
6.     <id name="sno" column="SNO">
7.       <generator class="increment"></generator>
8.     </id>
9.     <property name="name" column="NAME"></property>
10.    <property name="course" column="COURSE"></property>
11.    <property name="fee" column="FEE"></property>
12.
13.    <component name="address" class="com.sekharit.hibernate.entity.Address">
14.      <property name="houseNumber" column="HOUSE_NUMBER"></property>
15.      <property name="street" column="STREET"></property>
16.      <property name="city" column="CITY"></property>
17.    </component>
18.
19.  </class>
20. </hibernate-mapping>
```

CreateDAO1.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4. import org.hibernate.Transaction;
5.
6. import com.sekharit.hibernate.entity.Address;
7. import com.sekharit.hibernate.entity.Employee;
8. import com.sekharit.hibernate.util.SessionUtil;
9.
10. public class CreateDAO1 {
11.
12.     public static void main(String[] args) {
13.         Session session = SessionUtil.getSession();
14.         Transaction transaction = session.beginTransaction();
15.
16.         Address address = new Address();
17.         address.setHouseNumber("153-C");
18.         address.setStreet("Kesav Coloney");
19.         address.setCity("Gunipalli");
20.
21.         Employee employee = new Employee();
22.         employee.setAddress(address);
23.         employee.setName("yellareddy");
24.         employee.setDesignation("engineer");
25.         employee.setSalary(32000);
26.
27.         session.save(employee);
28.
29.         transaction.commit();
30.         session.close();
```

```
31.      }
32. }
```

Output Table:

EMP					
ENO	NAME	DESIG	SAL	HOUSE_NUMBER	STREET
CITY					
1	yellareddy	engineer	32000	153-C	Kesav Colony
					Gunipalli

RetrieveDAO1.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4. import org.hibernate.Transaction;
5.
6. import com.sekharit.hibernate.entity.Employee;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class RetrieveDAO1 {
10.
11.     public static void main(String[] args) {
12.         Session session = SessionUtil.getSession();
13.         Transaction transaction = session.beginTransaction();
14.
15.         Employee employee = (Employee) session.get(Employee.class, 1);
16.
17.         sop("Employee details are .....");
18.         sop("Ename :" + employee.getName());
19.         sop("salary :" + employee.getSalary());
20.         sop("Desig :" + employee.getDesignation());
21.         sop("HouseNo :" + employee.getAddress().getHouseNumber());
22.         sop("Street :" + employee.getAddress().getStreet());
23.         sop("City :" + employee.getAddress().getCity());
24.
25.         transaction.commit();
26.         session.close();
27.     }
28.
29.     public static void sop(Object object) {
30.         System.out.println(object);
31.     }
32. }
```

OUTPUT:

```
Employee details are .....
Ename :yellareddy
salary :32000.0
Desig :engineer
HouseNo :153-C
Street :Kesav Colony
City :Gunipalli
```

UpdateDAO1.java

```
1. package com.sekharit.hibernate.dao;
```

```
2.  
3. import org.hibernate.Session;  
4. import org.hibernate.Transaction;  
5.  
6. import com.sekharit.hibernate.entity.Employee;  
7. import com.sekharit.hibernate.util.SessionUtil;  
8.  
9. public class UpdateDAO1 {  
10.  
11.     public static void main(String[] args) {  
12.         Session session = SessionUtil.getSession();  
13.         Transaction transaction = session.beginTransaction();  
14.  
15.         Employee employee = (Employee) session.get(Employee.class, 1);  
16.         employee.setSalary(5698.0);  
17.         employee.getAddress().setHouseNumber("PIN-45-1");  
18.  
19.         transaction.commit();  
20.         session.close();  
21.     }  
22. }
```

Output Table:

EMP					
ENO	NAME	DESIG	SAL	HOUSE_NUMBER	STREET
1	yellareddy	engineer	5698	PIN-45-1	Kesav Coloney Gunipalli

DeleteDAO1.java

```
1. package com.sekharit.hibernate.dao;  
2.  
3. import org.hibernate.Session;  
4. import org.hibernate.Transaction;  
5.  
6. import com.sekharit.hibernate.entity.Employee;  
7. import com.sekharit.hibernate.util.SessionUtil;  
8.  
9. public class DeleteDAO1 {  
10.  
11.     public static void main(String[] args) {  
12.         Session session = SessionUtil.getSession();  
13.         Transaction transaction = session.beginTransaction();  
14.  
15.         Employee employee = (Employee) session.get(Employee.class, 1);  
16.         session.delete(employee);  
17.  
18.         transaction.commit();  
19.         session.close();  
20.     }  
21. }
```

Output Table:EMP

ENO	NAME	DESIG	SAL	HOUSE_NUMBER	STREET	CITY
1	yellareddy	engineer	5698	PIN-45-1	Kesav Coloney	Gunipalli

HQL, Criteria, Native-SQL and Named-Queries

- Till now we have done the operations on single object (single row), here we will see updates, deletes, selects on multiple rows of data (multiple objects) at a time.
- To write complicated queries **or** multiple where conditions **or** to write where conditions on other than primary key columns **or** to work on multiple records we use these features.
- To write SELECT, UPDATE, DELETE queries we can use **HQL**.
- In object oriented manner if we want to write the queries we can use **Criteria**. It supports only SELECT queries.
- To write very complicated queries like unions, joins...etc or which are not supported by the HQL and criteria, for them we use **Native SQL**. These are database dependent queries. Maximum try to avoid using Native SQL.
- If we write queries (HQL or Native SQL) declaratively(in mapping file/in annotations) then we can say those are **Named Queries**.

HQL (Hibernate Query Language)

- An object oriented form of SQL is called HQL
- HQL syntax is very much similar to SQL syntax
- HQL queries are formed by using Entities and their properties, whereas SQL queries are formed by using Tables and their columns.

Example:

SQL: SELECT ACCNO, NAME, BALANCE FROM ACCOUNT

HQL: SELECT accountId, name, balance fRoM Account

- In HQL query keywords (SELeCt, fRoM, AnD, oR, WhERe...etc) are not case sensitive whereas Entity name and Entity properties names are case sensitive(SQL queries are not case sensitive)
- HQL queries are database independent queries (Becoz HQL queries internally converted into database specific SQL queries using Dialect class).
- We can perform SELECT, UPDATE, DELETE ...etc, but we can't perform INSERT using HQL
- Both positional and named parameters concepts are supported by hibernate.

NOTE: Named parameters are advisable in projects. Because if any changes occur in the positions of the parameters no need to do changes in parameters setting.

- We can execute aggregate functions (MAX (), MIN (), AVG ()...etc) using HQL.
- supports polymorphic queries
- Return result as object
- INSERT queries we can't execute using HQL. But if we are selecting the complete data from one table to another table(backup table creation) that type of insert query is supported by HQL.

SQL: CREATE TABLE ACCOUNT_BACKUP AS (

 SELECT * FROM ACCOUNT)

- This feature is not at all advisable. Because database side creating backup table is very simple. Generally we no need to create backup tables from hibernate side
- Support for advance features like pagination, fetch join with dynamic profiling, Inner/outer/full joins, Cartesian products, Projection, Aggregation (max, avg) and grouping, Ordering, Sub queries and SQL function calls.

Steps to work with HQL

Step 1: Write the HQL query as per requirement

Step 2: Create a **org.hibernate.Query** object by passing HQL query

```
Query query = session.createQuery(hqlQuery);
```

Step 3: If the query contains parameters, set those values

```
query.setParameter(index, value);
```

Step 4: Execute query object

```
query.list(); [for executing SELECT queries]
```

```
query.executeUpdate(); [for executing UPDATE/DELETE queries]
```

org.hibernate.Query

- **Query** is an interface, **QueryImpl** is the implemented class
- It is an object oriented representation of Hibernate Query

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

public int executeUpdate(): is used to execute the update or delete query.
 public List list(): returns the result of the select query as a list.
 public Query setFirstResult(int rowno): specifies the row number from where record will be retrieved.
 public Query setMaxResult(int rowno): specifies the no. of records to be retrieved from the relation (table).
 public Query setParameter(int position, Object value): it sets the value to the JDBC style query parameter.
 public Query setParameter(String name, Object value): it sets the value to a named query parameter.

To work with all HQL examples I am using the following Entity

Account.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.Table;
9.
10. import org.hibernate.annotations.GenericGenerator;
11.
12. @Entity
13. @Table(name = "ACCOUNT")
14. public class Account {
15.     @Id
16.     @GenericGenerator(name = "myGen", strategy = "increment")
17.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGen")
18.     @Column(name = "ACCNO")
19.     private int accountId;
20.     @Column(name = "NAME")
21.     private String name;
22.     @Column(name = "BALANCE")
23.     private double balance;
24.
25.     public Account() {
26.
27. }
28.
29.     public Account(int accountId, String name, double balance) {
30.         This. accountId = accountId;
31.         this.name = name;
32.         this.balance = balance;
33.     }
34.
35.     // setters & getters
36.
37.     @Override
38.     public String toString() {
39.         return "Account [accountId=" + accountId + ", name=" + name
40.                         + ", balance=" + balance + "]";
41.     }
42.
43. }
  
```

UPDATE using HQL

SQL: UPDATE ACCOUNT SET NAME='sekhar_new' WHERE ACCNO=1001

- ⇒ By using **session.update()** method also we can achieve above functionality. Because single where statement and the condition is on 'id' column and the condition is equality condition.

SQL: UPDATE ACCOUNT SET NAME='sekhar_new' WHERE BALANCE=2800;

- ⇒ By using **session.update()** we can't achieve the above functionality even it contains the single condition. Bcz the condition is not on the 'id' column. Here exactly the need of HQL is realized.

Example:

SQL: UPDATE ACCOUNT SET NAME='sekhar_new' WHERE BALANCE=2800;

HQL: UPDATE Account a SET a.name='sekhar_new' WHERE a.balance=2800;

Code block:

```
1. String hqlQuery="UPDATE Account a SET a.name='sekhar_new' WHERE a.balance=2800";
2. Query query=session.createQuery(hqlQuery);
3. int rowsUpdated=query.executeUpdate();
4. sop("No.of rows updated :"+rowsUpdated);
```

DELETE using HQL

Example:

SQL: DELETE FROM ACCOUNT WHERE NAME LIKE '%sekhar%' AND BALANCE > 4000

HQL: DELETE FROM Account a WHERE a.name LIKE 's%' AND a.balance>4000

Code block:

```
1. String hqlQuery = " DELETE FROM Account a WHERE a.name LIKE 's%' AND
   a.balance>4000";
2. Query query = session.createQuery(hqlQuery);
3. int rowDeleted = query.executeUpdate();
4. sop("no.of rows deleted :"+ rowDeleted);
```

SELECT using HQL

The SELECT Hibernate query is of the form:

From <<package>>. <<Entity Class Name>>

From <<Entity Class Name>>
From <<Entity Class Name>> as <<alias name>>
From <<Entity Class Name>> <<alias name>>

Example:

SQL: SELECT * FROM ACCOUNT

HQL: FROM Account

OR

FROM Account a

OR

SELECT a FROM Account a

Code block:

```
1. String hql = "SELECT a FROM Account a";  
2. Query query = session.createQuery(hql);  
3. List<Account> accounts = query.list();  
4. sop(accounts);
```

Example:

SQL: SELECT * FROM ACCOUNT WHERE NAME LIKE 'sekhar%' AND BALANCE > 2800

HQL: SELECT a FROM Account a WHERE a.name LIKE 'sekhar%' AND a.balance > 2800

Code block:

```
1. String hql = "SELECT a FROM Account a WHERE a.name LIKE 'sekhar%' AND  
a.balance > 2000";  
2. Query query = session.createQuery(hql);  
3. List<Account> accounts = query.list();  
4. sop(accounts);
```

OR

```
1. String hql = "SELECT a FROM Account a WHERE a.name LIKE 'sekhar%' AND  
a.balance > 2000";  
2. List<Account> accounts = session.createQuery(hql).list();  
3. sop(accounts);
```

Parameter binding examples

Without parameter binding, you have to concatenate the parameter String like this (bad code) :

```
1. String hql = "from Account a where a.accountId = " + accountId;
2. List result = session.createQuery(hql).list();
```

Passing an unchecked value from user input to the database will raise security concern, because it can easily get hacked by SQL injection. You have to avoid the above bad code and use parameter binding instead.

Hibernate parameter binding

There are two ways to parameter binding : named parameters or positional.

1. Named parameters

This is the most common and user friendly way. It uses colon followed by a parameter name (:example) to define a named parameter. See examples...

Example 1 – setParameter

The **setParameter** is smart enough to discover the parameter data type for you.

```
1. String hql = " from Account a where a.accountId = :accountId ";
2. List<Account> accounts = session.createQuery(hql)
3.                     .setParameter("accountId", 9001)
4.                     .list();
```

Example 2 – setInteger

You can use **setInteger** to tell Hibernate this parameter data type is Integer.

```
1. String hql = " from Account a where a.accountId = :accountId";
2. List<Account> accounts = session.createQuery(hql)
3.                     .setInteger("accountId", 9001)
4.                     .list();
```

Example 3 – setProperties(bean)

You can pass an object into the parameter binding. Hibernate will automatically check the object's properties and match with the colon parameter.

```
1. Account account = new Account();
2. account.setAccountId(9001);
3. String hql = " from Account a where a.accountId = :accountId ";
4. List<Account> accounts = session.createQuery(hql)
5.                     .setProperties(account)
6.                     .list();
```

Example 4 – setProperties(map)

You can pass map into the parameter binding. Hibernate will automatic check the map keys and match with the colon parameter.

```

1. Map<String, Object> map = new HashMap<String, Object>();
2. map.put("accountId", 9001);
3. String hql = " from Account a where a.accountId = :accountId ";
4. List<Account> accounts = session.createQuery(hql)
5.           .setProperties(map)
6.           .list();

```

Example Code block :

```

1. String hql = "SELECT a FROM Account a
2.           WHERE a.name LIKE :name AND a.balance > :balance";
3. Query query = session.createQuery(hql);
4. query.setParameter("name", "%sekhar%");
5. query.setParameter("balance", 2000.0);
6. List<Account> accounts = query.list();
7. sop(accounts);

```

OR

```

1. String hql = "SELECT a FROM Account a
2.           WHERE a.name LIKE :name AND a.balance > :balance";
3. List<Account> accounts = session.createQuery(hql)
4.           .setParameter("name", "%sekhar%")
5.           .setParameter("balance", 2000.0)
6.           .list();

```

NOTE: In future, even if we are removing name condition no need to change other conditions.

Positional parameters

It's use question mark (?) to define a named parameter, and you have to set your parameter according to the position sequence. See example...

```

1. String hql = " Account a where a.accountId =? and a.name=?";
2. List result = session.createQuery(hql)
3.           .setParameter(0, 9001)
4.           .setParameter(1, "%sekhar%")
5.           .list();

```

OR

```

1. String hql = " Account a where a.accountId =? and a.name=?";
2. List result = session.createQuery(hql)
3.           .setInteger(0, 9001)
4.           .setString(1, "%sekhar%")
5.           .list();

```

Example Code block :

```

1. String hql = "SELECT a FROM Account a WHERE a.name LIKE ? AND a.balance > ?";
2.         Query query = session.createQuery(hql);
3.         query.setParameter(0, "%sekhar%");
4.         query.setParameter(1, 2000.0);
5.         List<Account> accounts = query.list();
6.         sop(accounts);

```

OR

```

1.     List<Account> accounts = session.createQuery(hql)
2.                     .setString(0, "%sekhar%")
3.                     .setDouble(1, 2000.0)
4.                     .list();

```

- ⇒ This approach is not support the **setProperties** method. In addition In future, if we remove any condition in between the query then parameter setting code will get affected.

Pros/Cons

- Position-based faster on executing variable substitution
- Name-based doesn't require code changes if a new parameter gets added in the middle of the statement

Conclusion

I would recommend always go for "**Named parameters**", as it's more easy to maintain.

NOTE: Even we can use both named parameters and positional parameters with in the same query. But we should write positional parameters first then named parameters. Inverse is not allowed. But mixing named and positional parameters is not at all advisable.

Referring to identifier property

There are 2 ways to refer to an entity's identifier property:

- o Using special property (lowercase) "**id**" may be used to reference the identifier property of an entity. *Rule is that the entity does not define a non-identifier property with name "id".*
- o Using identifier property name

NOTE: Starting in version 3.2.2, this has changed significantly. In previous versions, id *always* referred to the identifier property regardless of its actual name. After 3.2.2 versions non-identifier properties named id could never be referenced in Hibernate queries.

Example:

SQL: SELECT * FROM ACCOUNT WHERE ACCNO > 5001

HQL: SELECT a FROM Account a WHERE a.id > 5001

OR

SELECT a FROM Account a WHERE a.accountId > 5001

Code Block:

```

1. String hql = "SELECT a FROM Account a WHERE a.id>:accountId";
2. Query query = session.createQuery(hql);
3. query.setParameter("accountId", 5001);
4. List<Account> accounts = query.list();
5. sop(accounts);

```

Working with different return types of list() method

Example 1: full object selection (selecting all columns and all records of the table)

Example:

SQL: SELECT * FROM ACCOUNT

HQL: FROM Account

Code block:

```

1.     String hqlQuery="FROM Account";
2.     Query query=session.createQuery(hqlQuery) ;
3.     List<Account> accounts=query.list();
4.         for(Account account : accounts){
5.             sop("\nACCNO : "+account.getAccountId());
6.             sop("NAME : "+account.getName());
7.             sop("BALANCE : "+account.getBalance());
8.         }
9.
10.
11.
12.         for(int i =0 ;i<accounts.size(); i++){
13.             Account account = accounts.get(i);
14.             sop("\nACCNO : "+account.getAccountId());
15.             sop("NAME : "+account.getName());
16.             sop("BALANCE : "+account.getBalance());
17.         }
18.
19.         Iterator<Account> iterator = accounts.iterator();
20.         while(iterator.hasNext()){
21.             Account account =iterator.next();
22.             sop("\nACCNO : "+account.getAccountId());
23.             sop("NAME : "+account.getName());
24.             sop("BALANCE : "+account.getBalance());
25.         }
26.
27.         for(Iterator<Account> iterator = accounts.iterator();

```

```

28.                                     iterator.hasNext());
29.             Account account = iterator.next();
30.             sop("\nACCNO : "+account.getAccountId());
31.             sop("NAME : "+account.getName());
32.             sop("BALANCE : "+account.getBalance());
33.         }

```

- When we select all columns and all records from the ACCOUNT table using **list()** method it returns **java.util.List** object which contains **Account** object per each record of the **ACCOUNT** table.

i.e. 'n' records - '1' List Object

'n' records - 'n' Account objects available in List.

Example 2: partial object selection (selecting multiple columns/values)

Case 1: selecting multiple columns

Example:

SQL: SELECT NAME, BALANCE FROM ACCOUNT

HQL: SELECT a.name, a.balance FROM Account a

Note: It is mandatory to have alias name for complicated queries like inner queries, correlated nested queries...etc.

Code block:

```

1. String hqlQuery = "SELECT acc.name, acc.balance FROM Account acc";
2. Query query = session.createQuery(hqlQuery);
3. List<Object[]> accounts = query.list();
4. for (Object[] record : accounts) {
5.     for (Object column : record) {
6.         System.out.print(column+"\t");
7.     }
8.     System.out.println("\n.....\n");
9. }

```

- If we retrieve multiple columns by specifying their names **list()** method returns **java.util.List** object which contains one **Object[]** arrays per each record. Then each object array contains selected property type of values.

i.e. 'n' records - '1' List Object

'n' records - List contains 'n' **Object[]** arrays

'm' columns - 'm' objects of selected property types present in each **Object[]** array.

Case 2 : Selecting multiple values

Example:

SQL: SELECT MAX(BALANCE), MIN(BALANCE) FROM ACCOUNT

HQL: SELECT MAX(a.balance), MIN(a.balance) FROM Account a

Code block:

```

1. String hql = "SELECT MAX(a.balance), MIN(a.balance) FROM Account a";
2.         Query query = session.createQuery(hql);
3.         List<Object[]> list = query.list();
4.
5.         for(Object[] values : list){
6.             sop("\nMax : "+values[0]);
7.             sop("Min : "+values[1]);
8.         }

```

- HERE **list** contains one **Object[]** array, The **Object[]** array size is depends on no.of aggregate functions we are selecting.

Example3: partial object selection (Selecting single column/vlaue)

Case 1 : Selecting single column

Example:

SQL: SELECT name FROM ACCOUNT

HQL: SELECT a.name FROM Account a

Code block:

```

1. String hqlQuery = "SELECT a.name FROM Account a";
2.         Query query = session.createQuery(hqlQuery);
3.         List<String> names = query.list();
4.         for (String name : names) {
5.             System.out.print(name);
6.             System.out.println("\n.....\n");
7.         }

```

- Here **List** contains one Object per each record. Type of elements in the **List** depends on selected property type.

i.e. 'n' records - '1' **List** Object

'n' records – **List** contains 'n' Objects of selected property type.

Case 2: Selection single value

Example:

SQL: SELECT AVG(BALANCE) FROM ACCOUNT

HQL: SELECT AVG(a.balance) FROM Account a

Code block:

```

1. String hql = "SELECT AVG(a.balance) FROM Account a";
2.         Query query = session.createQuery(hql);
3.         List<Double> list = query.list();
4.         sop("Average : "+list.get(0));

```

- Here the **list** contains only one object. The type of the object in **list**, depends on the aggregate function return type.

Aggregate functions

Functions that operate against groups of resulting records. The supported aggregate functions are:

- avg(...), sum(...), min(...), max(...)
- o count(*)
- o count(...), count(distinct ...), count(all...)

Example:

SQL: SELECT MIN(BALANCE), MAX(BALANCE), AVG(BALANCE) FROM ACCOUNT

HQL: SELECT MIN(a.balance), MAX(a.balance), AVG(a.balance) FROM Account a

Code block:

```

1. String hqlQuery ="SELECT MIN(a.balance), MAX(a.balance), AVG(a.balance) FROM
   Account a ";
2. Query query = session.createQuery(hqlQuery);
3. List<Object[]> aggrs = query.list();
4. for (Object[] aggrObject : aggrs) {
5.     sop("Min:" + aggrObject[0]);
6.     sop("Max:" + aggrObject[1]);
7.     sop("Avg:" + aggrObject[2]);
8. }

```

Aggregate function with “GROUP BY” clause

Example 1: Group by single column

SQL: SELECT NAME, AVG(BALANCE) FROM ACCOUNT GROUP BY NAME

HQL: SELECT a.name, AVG(a.balance) FROM Account a GROUP BY a.name

Code block:

```

1. String hql = "SELECT a.name, AVG(a.balance) FROM Account a GROUP BY a.name";
2.         Query query = session.createQuery(hql);
3.         List<Object[]> list = query.list();
4.
5.         for(Object[] group: list){
6.             sop("\nName : "+group[0]);
7.             sop("Avg Balance : "+group[1]);
8.         }

```

- Here **List** object contains one **Object[]** array per each group. Type of type elements in the **Object[]** array depends on the aggregate function return type, selected property types.

Example 2: Group by multiple columns

SQL: SELECT name,balance,max(balance), min(balance), avg(balance) FROM ACCOUNT GROUP BY BALANCE, NAME

HQL: SELECT a.name, a.balance, max(a.balance), min(a.balance), avg(a.balance) FROM Account a GROUP BY a.balance, a.name

Code block:

```

1. String hql = "SELECT a.name, a.balance, max(a.balance), min(a.balance), avg(a.balance)
2.                         FROM Account a GROUP BY a.balance, a.name";
3.         Query query = session.createQuery(hql);
4.         List<Object[]> list = query.list();
5.
6.         for(Object[] group: list){
7.             sop("\nName : "+group[0]);
8.             sop("Balance : "+group[1]);
9.             sop("Max Balance : "+group[2]);
10.                sop("Min Balance : "+group[3]);
11.                sop("Avg Balance : "+group[4]);
12.         }

```

Aggregate function with "GROUP BY" clause and "HAVING" clause**Example:**

SQL: SELECT NAME, AVG(BAL) FROM ACCOUNT GROUP BY NAME HAVING NAME LIKE '%sekhar%'

HQL: SELECT a.name, AVG(a.balance) FROM Account a GROUP BY a.name HAVING a.name LIKE '%sekhar%'

Code block:

```

1. String hql = "SELECT a.name, AVG(a.balance) FROM Account a
2.           GROUP BY a.name HAVING a.name LIKE '%sekhar%'";
3.           Query query = session.createQuery(hql);
4.           List<Object[]> list = query.list();
5.
6.           for(Object[] group: list){
7.               sop("\nName : "+group[0]);
8.               sop("Avg Balance : "+group[1]);
9.           }

```

"ORDER BY" clause with "ASC" and "DESC"

Example 1: single column on order by clause

SQL: SELECT * FROM ACCOUNTA ORDER BY BALANCE

HQL: FROM Account a ORDER BY a.balance

Code block:

```

1. String hql = "FROM Account a ORDER BY a.balance";
2.           Query query = session.createQuery(hql);
3.           List<Account> list = query.list();
4.
5.           for(Account account : list){
6.               sop("\nAccountId : "+account.getAccountId());
7.               sop("Name : "+account.getName());
8.               sop("Balance : "+account.getBalance());
9.           }

```

NOTE: ORDER BY clause by default it will take Ascending order. If we want to specify the order explicitly, we can write the query as follows.

HQL: FROM Account a ORDER BY a.balance ASC (Ascending order)

HQL: FROM Account a ORDER BY a.balance DESC (Descending order)

Example 2: Multiple columns on order by clause

SQL: SELECT * FROM ACCOUNT ORDER BY BALANCE, NAME

HQL: select * FROM Account a ORDER BY a.balance, a.name

Code block:

```

1. String hql = "FROM Account a ORDER BY a.balance, a.name";
2.         Query query = session.createQuery(hql);
3.         List<Account> list = query.list();
4.
5.         for(Account account : list){
6.             sop("\nAccountId : "+account.getAccountId());
7.             sop("Name : "+account.getName());
8.             sop("Balance : "+account.getBalance());
9.         }

```

Example 3: Multiple columns on order by clause, with different ordering**SQL:** SELECT * FROM ACCOUNT ORDER BY BALANCE DESC, NAME ASC, ACCNO**HQL:** FROM Account a ORDER BY a.balance DESC, a.name ASC, a.accountId**Code block:**

```

1. String hql = "FROM Account a ORDER BY a.balance DESC, a.name ASC, a.accountId";
2.         Query query = session.createQuery(hql);
3.         List<Account> list = query.list();
4.
5.         for(Account account : list){
6.             sop("\nAccountId : "+account.getAccountId());
7.             sop("Name : "+account.getName());
8.             sop("Balance : "+account.getBalance());
9.         }

```

'BETWEEN' Clause**SQL:** SELECT * FROM ACCOUNT WHERE BALANCE BETWEEN 2000 AND 8000**HQL:** FROM Account a WHERE a.balance between :lowerBound and :upperBound

The negated forms can be written as follows:

SQL: SELECT * FROM ACCOUNT WHERE BALANCE NOT BETWEEN 2000 AND 8000**HQL:** FROM Account a WHERE a.balance not between :lowerBound and :upperBound**Code block:**

```

1. String hql = "FROM Account a WHERE a.balance between :lowerBound and :upperBound";
2.         Query query = session.createQuery(hql);
3.         query.setParameter("lowerBound", 2000.0);
4.         query.setParameter("upperBound", 8000.0);
5.         List<Account> list = query.list();
6.
7.         for (Account account : list) {
8.             sop("\nAccountId : " + account.getAccountId());
9.             sop("Name : " + account.getName());

```

```

10.           sop("Balance : " + account.getBalance());
11.       }

```

'IN' Clause

SQL: SELECT * FROM ACCOUNT WHERE NAME IN ('sekhar', 'kesavareddy','somu')

HQL: FROM Account a WHERE a.name in (:listOfvalues)

The negated forms can be written as follows:

SQL: SELECT * FROM ACCOUNT WHERE NAME NOT IN('sekhar', 'kesavareddy','somu')

HQL: FROM Account a WHERE a.name not in (:listOfvalues)

Code block:

```

1. String hql = "FROM Account a WHERE a.name in (:listOfvalues)";
2.         Query query = session.createQuery(hql);
3.         List<String> listOfvalues= new ArrayList<String>();
4.         listOfvalues.add("sekhar");
5.         listOfvalues.add("kesavareddy");
6.         listOfvalues.add("somu");
7.         query.setParameterList("listOfvalues", listOfvalues);
8.         List<Account> list = query.list();
9.
10.        for (Account account : list) {
11.            sop("\nAccountId : " + account.getAccountId());
12.            sop("Name : " + account.getName());
13.            sop("Balance : " + account.getBalance());
14.        }

```

Code block:

```

1. String hql = "FROM Account a WHERE a.name in (:listOfvalues)";
2.         Query query = session.createQuery(hql);
3.         Object[] listOfvalues = new Object[3];
4.         listOfvalues[0] = "sekhar";
5.         listOfvalues[1] = "kesavareddy";
6.         listOfvalues[2] = "somu";
7.         query.setParameterList("listOfvalues", listOfvalues);
8.         List<Account> list = query.list();
9.
10.        for (Account account : list) {
11.            sop("\nAccountId : " + account.getAccountId());
12.            sop("Name : " + account.getName());
13.            sop("Balance : " + account.getBalance());
14.        }

```

'LIKE' Clause

SQL: SELECT * FROM ACCOUNT WHERE NAME LIKE '%sekhar%'

HQL: FROM Account a WHERE a.name LIKE :name

The negated forms can be written as follows:

SQL: SELECT * FROM ACCOUNT WHERE NAME NOT LIKE '%sekhar%'

HQL: FROM Account a WHERE a.name NOT LIKE :name

Code block:

```

1. String hql = "FROM Account a WHERE a.name NOT LIKE :name";
2.         Query query = session.createQuery(hql);
3.         query.setParameter("name", "%sekhar%");
4.         List<Account> list = query.list();
5.
6.         for (Account account : list) {
7.             sop("\nAccountId : " + account.getAccountId());
8.             sop("Name : " + account.getName());
9.             sop("Balance : " + account.getBalance());
10.        }

```

Using SQL/Database Functions

SQL: SELECT UPPER(NAME), LOWER(NAME), SYSDATE FROM ACCOUNT

HQL: SELECT UPPER(a.name), LOWER(a.name), SYSDATE FROM Account a

Code block:

```

1. String hql = "SELECT UPPER(a.name), LOWER(a.name), SYSDATE FROM Account a";
2.         Query query = session.createQuery(hql);
3.         List<Object[]> list = query.list();
4.
5.         for (Object[] cols : list) {
6.             sop("\nUpper Name : " + cols[0]);
7.             sop("Lower Name : " + cols[1]);
8.             sop("Sysdate : " + cols[2]);
9.         }

```

uniqueResult()

- ⇒ If we are expecting only one record as the result of query execution then we will call **uniqueResult()** method on **Query** object. If query returns more than one record this method throws **NonUniqueResultException**.

Table : ACCOUNT

ACCNO	NAME	BALANCE
5001	sekhar	8200
5002	yellareddy	9000
5003	sekhar	8900

Code Block:

```

1. String hql = "FROM Account a WHERE a.name=:name";
2.         Query query = session.createQuery(hql);
3.         query.setParameter("name", "sekhar");
4.         Account account = (Account) query.uniqueResult();
5.         sop("Account details are...\"");
6.         sop("\nAccountId : " + account.getAccountId());
7.         sop("Name : " + account.getName());
8.         sop("Balance : " + account.getBalance());

```

- ⇒ If the above Query returns more than one record we get Exception :
- org.hibernate.NonUniqueResultException:** query did not return a unique result: 2
- ⇒ If the query returns only one record **uniqueResults()** method execute the query and returns the resulted object

Code Block:

```

1. String hql = "FROM Account a WHERE a.accountId=:accountId ";
2.         Query query = session.createQuery(hql);
3.         query.setParameter("accountId", 5001);
4.         Account account = (Account) query.uniqueResult();
5.         sop("Account details are...\"");
6.         sop("\nAccountId : " + account.getAccountId());
7.         sop("Name : " + account.getName());
8.         sop("Balance : " + account.getBalance());

```

Code Block:

```

1.     String hql = "SELECT a.name FROM Account a WHERE a.accountId=:accountId";
2.     Query query = session.createQuery(hql);
3.     query.setParameter("accountId", 5001);
4.     String name = (String) query.uniqueResult();
5.     sop("Name :" + name);

```

Code Block:

```

1. String hql = "SELECT a.name, a.balance FROM Account a WHERE a.accountId=:accountId";
2.         Query query = session.createQuery(hql);
3.         query.setParameter("accountId", 5001);
4.         Object[] cols = (Object[]) query.uniqueResult();
5.         sop("Name :" + cols[0]);
6.         sop("Balance :" + cols[1]);

```

- ⇒ Generally we use this method where we should get only one record as query result.

iterate()

This method is also used to retrieve data from DB like list() method.

Code Block:

```

1. String hqlQuery = "from Account";
2.     Query query = session.createQuery(hqlQuery);
3.     Iterator<Account> accounts = query.iterate();
4.     Account account = null;
5.     while (accounts.hasNext()) {
6.         account = accounts.next();
7.         sop("AccountId:" + account.getAccountId());
8.         sop("Name:" + account.getName());
9.         sop("Balance:" + account.getBalance());
10.    }

```

NOTE: Enable “**show_sql**” property in **hibernate.cfg.xml** file and observe the console by executing above code block. You can find one select query per one record. That why it is not suggestible to use this method.

Differences between list() method and iterate method :

list()	iterate()
1.) Returns java.util.List object	1.) Returns java.util.Iterator object
2.) Query executed only once.	2.) Query executed per each record.
3.) If we want to process the results in presentation or service layer we use it.	3.) If we want to process the results in Data access layer itself then we use it.
4.) For more number of records selection it is suggested.	4.) For less number of records selection it is suggestible.

Pagination using Hibernate

In general when we are doing search, the search may return huge number of results. But at a time if we try to display those results in the webpage, it leads to the following problem.

- 1., Takes more time to load
2. Takes more memory to hold all results
3. Difficult to navigate over huge number of records

To solve this problem we can go for Pagination.

What is Pagination?

Pagination is the process of dividing (content) into multiple pages.

Example: In google search when we search, we will get the huge number of search results. But they will display only 10 links per page. And below the page they will display page numbers to navigate to other pages. This is nothing but Pagination.

Pagination can be implemented in two ways.

1. Front-End pagination
2. Back-end pagination

Front-End Pagination

In front-end pagination, we will hit the database only once, and get the results and store them in the memory(session/application scope). When user navigating from one page to another, we retrieve data from the memory but not hitting the database again and again.

With this style of pagination we can achieve easy navigation, fast loading of the data. But still we have the problems are there like

1. Takes more memory to hold all results
2. If data is updated in the database by other application, we can't get updated data, As we are taking the data from memory instead from database.

NOTE: This type of pagination is suggestible, if number of resulted records are limited and the data is constant data.

Back-End Pagination

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

With this style of pagination we can achieve easy navigation, fast loading of the data, takes memory to hold the result data And it will get always updated data. But it gives less performance when compared with front-end pagination.

NOTE: This type of pagination is suggestible, if number of resulted records are un-limited and the data is not constant data.

For back-end pagination, we need to write a query in such a way that, it returns only the requested page details.

To implement this queries different databases given different style of syntax.

For example : If want the results from 21st record to 30 record

Oracle :

```

select
  *
from
  ( select
    row_.*,
    rownum rownum_
  from
    ( select
      account0_.ACCNO as ACCNO1_,
      account0_.BALANCE as BALANCE1_,
      account0_.NAME as NAME1_
    from
      ACCOUNT account0_ ) row_
  where
    rownum <= ?
  )
  where
    rownum_ > ?
  
```

MYSQL:

```
select .... from ... order by .... limit 21, 10
```

Like this database to database the query formation is different. To overcome this problem, we can use Hibernate support.

- ⇒ To apply pagination hibernate provides two methods in **org.hibernate.Query** interface.
 - **setFirstResult(int firstResult)**
 - **setMaxResults(int maxResults)**

setFirstResult(int firstResult) take the argument which specify from where records has to take.
setMaxResults(int maxResults) take the argument which specify how many records has to take.

Code Block:

```

1. String hqlQuery = "FROM Account a";
2. Query query = session.createQuery(hqlQuery);
3. query.setFirstResult(3); //Means start from 4th record
4. query.setMaxResults(5); //Means per each page 5 records to be displayed
5. List<Account> accounts = query.list();
6. for (Account account: accounts) {
7.     sop("AccountId :" + account.getAccountId());
8.     sop("Name:" + account.getName());
9.     sop("Balance:" + account.getBalance());
10. }
  
```

- ⇒ While using the pagination, In webpage we are responsible to display the page numbers, And when the user click on some page number, we need to send the request to server, and get that corresponding page results. To do all these things we need to implement some logic.

- ⇒ This logic on the webpage if want we can implement on our own or we can use some third party provided custom tags. One of such custom tags is <display> tag. This tag is given by Apache people.

Subqueries

- ⇒ For databases that support subselects, Hibernate supports subqueries within queries. A subquery must be surrounded by parentheses (often by an SQL aggregate function call). Even correlated subqueries (subqueries that refer to an alias in the outer query) are allowed.

SQL: SELECT * FROM ACCOUNT WHERE BALANCE>(SELECT AVG(BALANCE) FROM ACCOUNT)

HQL: FROM Account a WHERE a.balance > (SELECT AVG(b.balance) FROM Account b)

Code Block:

```

1. String hql = "FROM Account a WHERE a.balance > ( SELECT AVG(b.balance) FROM Account b )";
2.         Query query = session.createQuery(hql);
3.         List<Account> list = query.list();
4.
5.         for (Account account : list) {
6.             sop("\nAccountId : " + account.getAccountId());
7.             sop("Name : " + account.getName());
8.             sop("Balance : " + account.getBalance());
9.         }

```

SQL: SELECT ACNO, NAME, (SELECT AVG(BALANCE) FROM ACCOUNT), BALANCE FROM ACCOUNT

HQL: SELECT a.accountId, a.name, (SELECT AVG(b.balance) FROM Account b) AS avgBalance, a.balance FROM Account a

Code Block:

```

1. String hql = "SELECT a.accountId, a.name,
2.                 (SELECT AVG(b.balance) FROM Account b) AS avgBalance, a.balance FROM Account a";
3.         Query query = session.createQuery(hql);
4.         List<Object[]> list = query.list();
5.
6.         for (Object[] cols : list) {
7.             sop("\nAccountId : " + cols[0]);
8.             sop("Name : " + cols[1]);
9.             sop("avgBalance : " + cols[2]);
10.                sop("Balance : " + cols[2]);
11.        }

```

Polymorphic queries

A query like:

from Payment as pa

returns instances not only of **Payment**, but also of subclasses like **ChequePayment** and **CashPayment**. Hibernate queries can name *any* Java class or interface in the from clause. The query will return instances of all persistent classes that extend that class or implement the interface. The following query would return all persistent objects:

```
from java.lang.Object o
```

returns Object[]

HQL: SELECT a.name, a.balance FROM Account a

Code Block:

```
1. String hqlQuery = "SELECT a.name, a.balance FROM Account a";
2. Query query = session.createQuery(hqlQuery);
3. List<Object[]> accounts = query.list();
4. for (Object[] record : accounts) {
5.     for (Object column : record) {
6.         System.out.print(column+"\t");
7.     }
8.     System.out.println("\n.....\n");
9. }
```

- With this **query. list()** method returns **java.util.List** object and that list contains **Object[]** arrays.

returns java.util.List

HQL: SELECT new list(a.name, a.balance) FROM Account a

Code Block:

```
44.     String hql = "SELECT new list(a.name, a.balance) FROM Account a";
45.             Query query = session.createQuery(hql);
46.             List<List> lists = query.list();
47.
48.             for (List list : lists) {
49.                 sop("\nName : " + list.get(0));
50.                 sop("Balance : " + list.get(1));
51.             }
```

- With this **query. list()** method returns **java.util.List** object and that list contains again **java.util.List** objects one per each record. And size of each list object depends on no.of selected properties.

returns User defined object

HQL: SELECT new Account(a.accno, a.name, a.balance) FROM Account a

Code Block:

```
1. String hql = "SELECT new Account(a.name, a.balance) FROM Account a";
```

```

2.         Query query = session.createQuery(hql);
3.         List<Account> lists = query.list();
4.
5.         for (Account account : lists) {
6.             sop("\nAccountId : " + account.getAccountId());
7.             sop("Name : " + account.getName());
8.             sop("Balance : " + account.getBalance());
9.         }

```

- With this **query.list()** method returns **java.util.List** object and that list contains **Account** objects. No.of **Account** objects depends on no.of records.

Code Block:

```

1. String hql = "SELECT new com.sekharit.hibernate.entity.AccountForDisplay(a.name,
   a.balance) FROM Account a";
2.         Query query = session.createQuery(hql);
3.         List<AccountForDisplay> lists = query.list();
4.
5.         for (AccountForDisplay account : lists) {
6.             sop("Name : " + account.getName());
7.             sop("Balance : " + account.getBalance());
8.         }

```

returns Map object

HQL: SELECT new map(MAX(a.balance) AS maxBalance, MIN(a.balance) AS minBalance, COUNT(*) AS count) FROM Account a

Code Block:

```

1. String hql = "SELECT new map(MAX(a.balance) AS maxBalance, MIN(a.balance) AS
   minBalance, COUNT(*) AS count) FROM Account a";
2.         Query query = session.createQuery(hql);
3.         List list = query.list();
4.
5.         Map<String, Object> map = (Map<String, Object>) list.get(0);
6.         sop("count : "+map.get("count"));
7.         sop("maxBalance : "+map.get("maxBalance"));
8.         sop("minBalance : "+map.get("minBalance"));

```

Criteria

- ⇒ The **Criteria API** allows queries to be built at runtime without direct string manipulations.
- ⇒ **Criteria** is an API from hibernate to write Queries in Object oriented manner rather than SQL or HQL.
- ⇒ **Criteria** is also **database independent**, Because it internally generates HQL queries.
- ⇒ We can execute only **SELECT** statements using Criteria; we can't execute **UPDATE, DELETE** statements using Criteria.
- ⇒ Criteria is suitable for executing **dynamic queries**
- ⇒ Criteria API also include **Query by Example (QBE)** functionality for supplying example objects.
- ⇒ Criteria also includes **projection and aggregation** methods

NOTE: Queries expressed as **criteria** are less readable than queries expressed in **HQL**.

To work with all criteria examples I am using the following Entity

Account.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.Table;
9.
10. import org.hibernate.annotations.GenericGenerator;
11.
12. @Entity
13. @Table(name = "ACCOUNT")
14. public class Account {
15.     @Id
16.     @GenericGenerator(name = "myGen", strategy = "increment")
17.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGen")
18.     @Column(name = "ACCNO")
19.     private int accountId;
20.     @Column(name = "NAME")
21.     private String name;
22.     @Column(name = "BALANCE")
23.     private double balance;
24.
25.     public Account() {
26.
27. }
```

```
28.
29.     public Account(int accountID, String name, double balance) {
30.         This.accountID = accountID;
31.         this.name = name;
32.         this.balance = balance;
33.     }
34.
35.     public int getAccountID() {
36.         return accountID;
37.     }
38.
39.     public void setAccountID(int accountID) {
40.         this.accountID = accountID;
41.     }
42.
43.     public String getName() {
44.         return name;
45.     }
46.
47.     public void setName(String name) {
48.         this.name = name;
49.     }
50.
51.     public double getBalance() {
52.         return balance;
53.     }
54.
55.     public void setBalance(double balance) {
56.         this.balance = balance;
57.     }
58.
59.     @Override
60.     public String toString() {
61.         return "Account [accountID=" + accountID + ", name=" + name
62.                 + ", balance=" + balance + "]";
63.     }
64. }
65. }
```

Here's a case study to retrieve a list of Account objects, with optional search criteria – balance, name, accountID, order by accountID.

1. HQL example

In HQL, you need to compare whether this is the first criteria to append the 'where' syntax. It's work, but the long codes are ugly, cumbersome and error-prone string concatenation may cause security concern like SQL injection.

Code block:

```
1. private static List<Account> getAccounts(Session session, Double balance,
2.                                         String name, Integer accountID) {
3.     boolean isFirst = true;
4. }
```

```

5.         StringBuilder query = new StringBuilder("FROM Account a ");
6.
7.         if (balance != null) {
8.             if (isFirst) {
9.                 query.append(" WHERE a.balance >= "+balance);
10.            } else {
11.                query.append(" AND a.balance >= "+balance);
12.            }
13.            isFirst = false;
14.        }
15.
16.        if (name != null) {
17.            if (isFirst) {
18.                query.append(" WHERE a.name LIKE '" + name + "'");
19.            } else {
20.                query.append(" AND a.name LIKE '" + name + "'");
21.            }
22.            isFirst = false;
23.        }
24.
25.        if (accountId != null) {
26.            if (isFirst) {
27.                query.append(" WHERE a.accountId >= " + accountId);
28.            } else {
29.                query.append(" AND a.accountId >= " + accountId);
30.            }
31.            isFirst = false;
32.        }
33.
34.        query.append(" ORDER BY a.accountId");
35.        Query result = session.createQuery(query.toString());
36.
37.        List<Account> accounts = result.list();
38.        System.out.println(accounts.size());
39.        return accounts;
40.    }

```

2. Criteria example

In Criteria, you do not need to compare whether this is the first criteria to append the 'where' syntax.. The line of code is reduce and everything is handled in a more elegant and object oriented way.

Code block:

```

1.     private static List<Account> getAccounts(Session session, Double balance,
2.                                                 String name, Integer accountId) {
3.         boolean isFirst = true;
4.
5.         Criteria criteria = session.createCriteria(Account.class);
6.         if (balance != null) {
7.             criteria.add(Restrictions.ge("balance", balance));
8.         }
9.         if (name != null) {
10.             criteria.add(Restrictions.like("name", name));
11.         }
12.         if (accountId != null) {
13.             criteria.add(Restrictions.ge("accountId", accountId));

```

```
14.        }
15.        criteria.addOrder(Order.asc("accountId"));
16.
17.        List<Account> accounts = criteria.list();
18.        sop(accounts);
19.        return accounts;
20.    }
```

Steps to work with Criteria API

Step 1: Create **org.hibernate.Criteria** Object

```
Criteria criteria = session.createCriteria(EntityClassName.class);
```

Step 2: Create **org.hibernate.criterion.Criterion** Object per each condition of the query and add to **Criteria** object.

```
Criterion criterion = Restrictions.eq("propertyName", propertyValue);
criteria.add(criterion);
```

Step 3: Execute **org.hibernate.Criteria** object (by calling **list()** method on **Criteria** object)

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

Q) Write and execute Criteria API code for the following SQL statement?

SQL: SELECT * FROM ACCOUNT WHERE NAME='sekhar' AND BALANCE>1800;

CRITERIA:

```
1. // SELECT * FROM ACCOUNT
2. Criteria criteria = session.createCriteria(Account.class);
3.
4. // NAME='sekhar'
5. Criterion nameCriterion = Restrictions.eq("name", "sekhar");
6.
7. // BALANCE>1800
8. Criterion balanceCriterion = Restrictions.gt("balance", 1800.0);
9.
10. // NAME='sekhar' AND BALANCE>1800
11. Criterion criterion = Restrictions.and(nameCriterion, balanceCriterion);
12.
13. // SELECT * FROM ACCOUNT WHERE NAME=' sekhar' AND BALANCE>1800;
14. criteria.add(criterion);
15.
16. List<Account> accounts = criteria.list();
17. for (Account account : accounts) {
18.     sop(account);
19. }
```

Q) How to implement the previous code using method chaining concept?

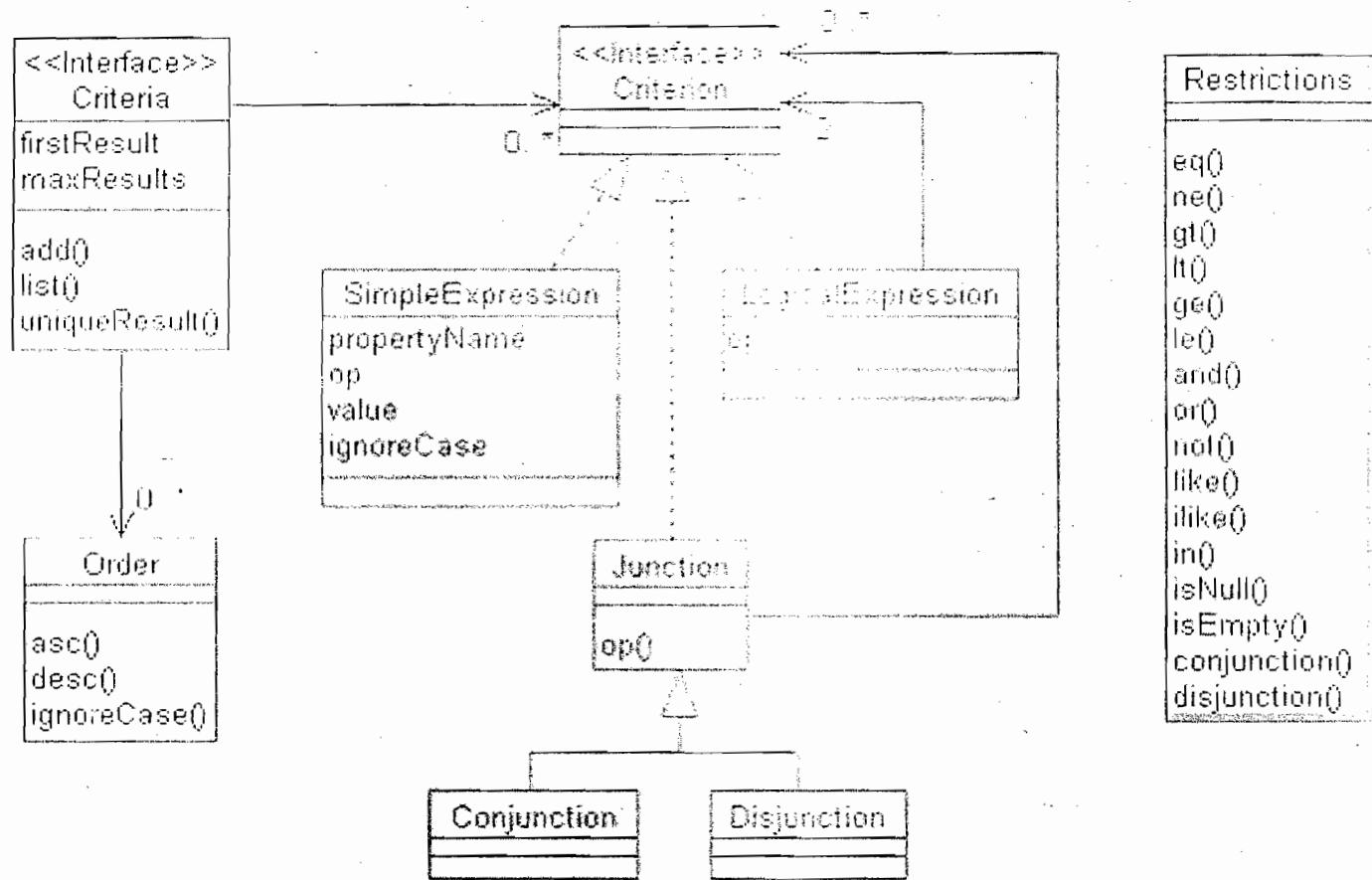
NOTE: As most Hibernate classes, the methods in Criteria return the **this** reference, so additional calls can be chained.

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.           .add(Restrictions.eq("name", "sekhar"))
3.           .add(Restrictions.gt("balance", 1800.0))
4.           .list();
5.
6.           for (Account account : accounts) {
7.               sop(account);
8.           }
  
```

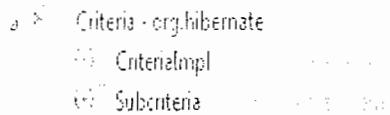
Q) What do you know about `org.hibernate.Criteria`, `org.hibernate.criterion.Criterion`, `org.hibernate.criterion.Restrictions` and `org.hibernate.criterion.Expression`?

- The following UML diagram summarizes the fundamentals of the Criteria API:



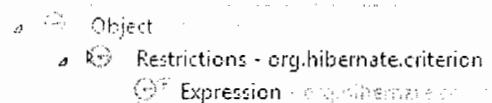
- **Criteria** is an interface used to represent query in Object oriented format. It contains methods to add **Criterion** objects, **Order** objects, **Projection** objects, pagination methods ...etc. **CriteriaImpl** is the implementation class of **Criteria**.

Type hierarchy of 'org.hibernate.Criteria':



- **Restrictions** is class which has methods to define conditions. These class methods return different **XXXExpression** classes which implements **Criterion** interface. So **Restrictions** is a static factory for **Criterion** instances.
- **Expression** class is child class of **Restrictions**. It defines methods to implement native SQL. As latest versions of Hibernate provides separate **Native SQL API**, this class made as deprecated.

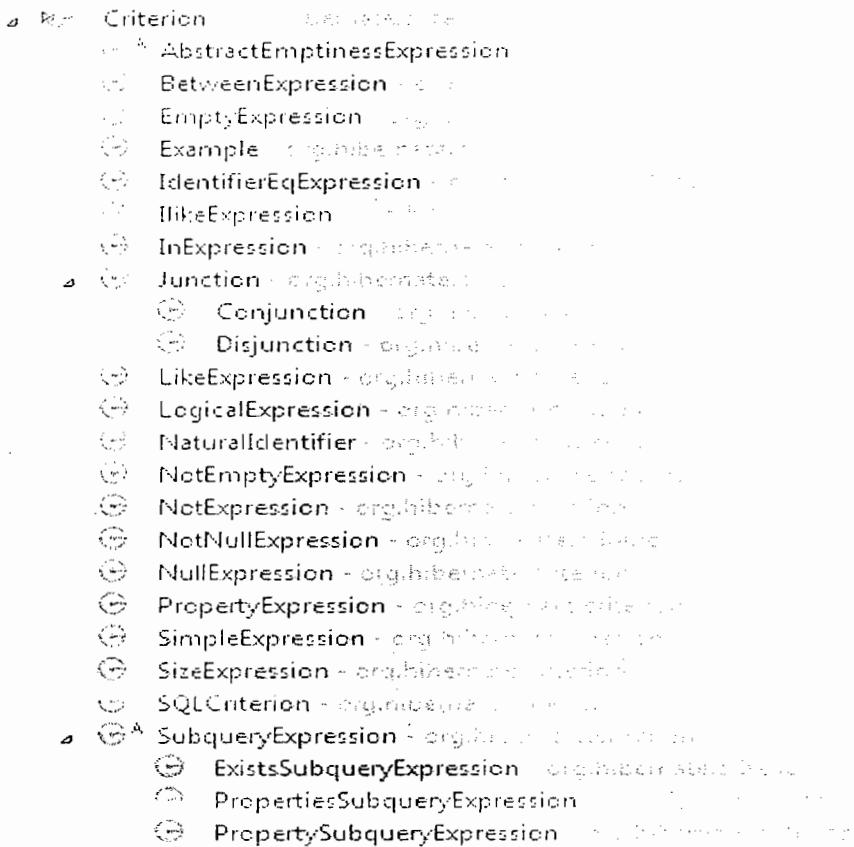
Type hierarchy of 'org.hibernate.criterion.Restrictions':



- **Criterion** is an interface which is used to represent one condition in object oriented manner. By calling methods on **Restrictions** we will get this object. There are different types of **XXXExpression** classes which are implementing **Criterion** interface. But all those implementation are Abstracted with **Criterion** interface

Expression class	Restrictions method	SQL operator/function
SimpleExpression	eq(prop, val)	=
SimpleExpression	ne(prop, val)	!=, <>
SimpleExpression	like(prop, val) [case-sensitive]	LIKE
SimpleExpression	gt(prop, val)	>
SimpleExpression	lt(prop, val)	<
SimpleExpression	ge(prop, val)	>=
SimpleExpression	le(prop, val)	<=
IdentifierExpression	idEq(val)	<Id-column> =
ILikeExpression	ilike(prop, val) [case-in-sensitive]	LIKE
BetweenExptession	between(prop, val, val)	BETWEEN <lowerLimit> AND <upperLimit>
InExpression	in(prop, val[])	IN(val1, val2....)
NullExpression	isNull(prop)	<column> IS NULL
NullExpression	isNotNull(prop)	<column> IS NOT NULL
LogicalExpression	and (criterion1, criterion2)	<expr1> AND <expr2>
LogicalExpression	or(criterion1, criterion2)	<expr1> OR <expr2>
NotExpression	not(criterioin)	!
EmptyExpression	isEmpty(prop)	EMPTY

Type hierarchy of 'org.hibernate.criterion.Criterion':



- **org.hibernate.criterion.Property** is a class available in hibernate API.
- **Property** is used to create **Criterion** objects(means creating conditions of the query).
- **Property** is just like **Restrictions** class. Both are meant for creating **Criterion** objects.
- If we are using same property of entity, on multiple conditions it is better to go for **Property** rather than **Restrictions**.
- You can create a **Property** by calling **Property.forName("property-Name")**
- If we want to compare properties of entity, we can use the following methods of **Restrictions** class.
 - ⦿ **eqProperty(String, String) : PropertyExpression**
 - ⦿ **neProperty(String, String) : PropertyExpression**
 - ⦿ **ltProperty(String, String) : PropertyExpression**
 - ⦿ **leProperty(String, String) : PropertyExpression**
 - ⦿ **gtProperty(String, String) : PropertyExpression**
 - ⦿ **geProperty(String, String) : PropertyExpression**

If we want to compare properties we can use the following methods of **Property** class.

```

    . . .
    eqProperty(Property)
    neProperty(Property)
    leProperty(Property)
    geProperty(Property)
    ltProperty(Property)
    gtProperty(Property)
    eqProperty(String)
    neProperty(String)
    leProperty(String)
    geProperty(String)
    ltProperty(String)
    gtProperty(String)
    . . .
  
```

org.hibernate.criterion.Property class methods are...

```

Property
  . . .
  Property(String)
  between(Object, Object)
  in(Collection)
  in(Object[])
  like(Object)
  like(String, MatchMode)
  eq(Object)
  ne(Object)
  gt(Object)
  lt(Object)
  le(Object)
  ge(Object)
  eqProperty(Property)
  neProperty(Property)
  leProperty(Property)
  geProperty(Property)
  ltProperty(Property)
  gtProperty(Property)
  eqProperty(String)
  neProperty(String)
  leProperty(String)
  geProperty(String)
  ltProperty(String)
  gtProperty(String)
  isNull()
  isNotNull()
  isEmpty()
  isNotEmpty()
  count()
  max()
  min()
  avg()
  group()
  asc()
  desc()
  $ forName(String)
  
```

Q) Write and execute Criteria API code for the following SQL statement? Use **org.hibernate.criterion.Property** class to create conditions

SQL: SELECT * FROM ACCOUNT WHERE NAME='sekhar' AND BALANCE>1800;

CRITERIA:

```
1.      // SELECT * FROM ACCOUNT
```

```

2.     Criteria criteria = session.createCriteria(Account.class);
3.
4.     // NAME='sekhar'
5.     Property nameProperty = Property.forName("name");
6.     Criterion nameCriterion = nameProperty.eq("sekhar");
7.
8.     // BALANCE>1800
9.     Property balanceProperty = Property.forName("balance");
10.    Criterion balanceCriterion = balanceProperty.gt(1800.0);
11.
12.    // NAME='sekhar' AND BALANCE>1800;
13.
14.    Criterion criterion = Restrictions.and(nameCriterion, balanceCriterion);
15.
16.    // SELECT * FROM ACCOUNT WHERE NAME='sekhar' AND BALANCE>1800;
17.
18.    criteria.add(criterion);
19.
20.    List<Account> accounts = criteria.list();
21.    for (Account account : accounts) {
22.        sop(account);
23.    }

```

OR

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.                     .add(
3.                         Restrictions.and(
4.                             Property.forName("name").eq("sekhar"),
5.                             Property.forName("balance").gt(1800.0)))
6.                     .list();

```

Q.) How to apply Pagination in criteria ?

```

1. Criteria criteria = session.createCriteria(Account.class);
2.     criteria.setFirstResult(5);
3.     criteria.setMaxResults(5);
4.
5.     List<Account> accounts = criteria.list();
6.     for (Account account : accounts) {
7.         sop(account);
8.     }

```

Examples based on different conditions

Example: Id Equal Restriction

SQL: SELECT * FROM ACCOUNT WHERE ACCNO=5001

CRITERIA:

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.                     .add(Restrictions.eq("accountId", 5001))
3.                     .list();

```

OR

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.                     .add(Restrictions.idEq(5001))
3.                     .list();

```

OR

```
1. List<Account> accounts = session.createCriteria(Account.class)
2.                     .add(Property.forName("accountId").eq(5001))
3.                     .list();
```

Example: Sorting the result

SQL: SELECT * FROM ACCOUNT ORDER BY BALANCE DESC, NAME ASC, ACCNO DESC

CRITERIA:

```
1. List<Account> accounts = session.createCriteria(Account.class)
2.                     .addOrder(Property.forName("balance").desc())
3.                     .addOrder(Property.forName("name").asc())
4.                     .addOrder(Property.forName("accountId").desc())
5.                     .list();
```

OR

```
1. List<Account> accounts = session.createCriteria(Account.class)
2.                     .addOrder(Order.desc("balance"))
3.                     .addOrder(Order.asc("name"))
4.                     .addOrder(Order.desc("accountId"))
5.                     .list();
```

Example: Logical operators Restriction

SQL: SELECT * FROM ACCOUNT WHERE ACCNO >= 5001 AND ACCNO <= 5005

CRITERIA:

```
1. List<Account> accounts = session.createCriteria(Account.class)
2.                     .add(Restrictions.or(
3.                         Restrictions.ge("accountId", 5001),
4.                         Restrictions.le("accountId", 5005)))
5.                     .list();
```

SQL: SELECT * FROM ACCOUNT WHERE ACCNO > 5001 OR ACCNO < 5005

CRITERIA:

```
1. List<Account> accounts = session.createCriteria(Account.class)
2.                     .add(Restrictions.and(
3.                         Restrictions.gt("accountId", 5001),
4.                         Restrictions.lt("accountId", 5005)))
5.                     .list();
```

OR

```
1. List<Account> accounts = session.createCriteria(Account.class)
2.                     .add( Restrictions.gt("accountId", 5001))
3.                     .add( Restrictions.lt("accountId", 5005))
4.                     .list();
```

NOTE: If we directly add multiple criterion objects to Criteria, by default "AND" logical operator will be applied among the conditions of the generated query.

Example: Conjunction and Disjunction

Conditions can be combined using Restrictions.disjunction() and Restrictions.conjunction().

- No arguments are passed to either method.
- Criterion instances are then added to the conjunction or disjunction using their respective add methods.
- The conjunction or disjunction is then added to a Criteria instance and evaluated.

SQL: SELECT * FROM ACCOUNT WHERE ACCNO>5001 **AND** NAME ='sekhar' **AND** BALANCE>2500.0
AND BALANCE<9200.0

CRITERIA:

```

1.      Criteria criteria = session.createCriteria(Account.class);
2.
3.      Criterion c1 = Restrictions.gt("accountId", 5001);
4.      Criterion c2 = Restrictions.eq("name", "sekhar");
5.      Criterion c3 = Restrictions.gt("balance", 2500.0);
6.      Criterion c4 = Restrictions.lt("balance", 9200.0);
7.
8.      Criterion c5 = Restrictions.and(c1,c2);
9.      Criterion c6 = Restrictions.and(c3,c4);
10.
11.     Criterion c7 = Restrictions.and(c5,c6);
12.
13.     criteria.add(c7);
14.     List<Account> accounts = criteria.list();

```

OR

```

1.      Criteria criteria = session.createCriteria(Account.class);
2.
3.      Criterion c1 = Restrictions.gt("accountId", 5001);
4.      Criterion c2 = Restrictions.eq("name", "sekhar");
5.      Criterion c3 = Restrictions.gt("balance", 2500.0);
6.      Criterion c4 = Restrictions.lt("balance", 9200.0);
7.
8.      Conjunction conjunction = Restrictions.conjunction();
9.      conjunction.add(c1);
10.     conjunction.add(c2);
11.     conjunction.add(c3);
12.     conjunction.add(c4);
13.
14.     criteria.add(conjunction);
15.     List<Account> accounts = criteria.list();

```

OR

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.                     .add(Restrictions.conjunction()
3.                         .add(Restrictions.gt("accountId", 5001))
4.                         .add(Restrictions.eq("name", "sekhar"))
5.                         .add(Restrictions.gt("balance", 2500.0))
6.                         .add(Restrictions.lt("balance", 9200.0)))
7.                     .list());

```

SQL: SELECT * FROM ACCOUNT WHERE ACCNO>5001 OR NAME ='sekhar' OR BALANCE>2500.0 OR BALANCE<9200.0

CRITERIA:

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.                               .add(Restrictions.disjunction()
3.                                     .add(Restrictions.gt("accountId", 5001))
4.                                     .add(Restrictions.eq("name", "sekhar"))
5.                                     .add(Restrictions.gt("balance", 2500.0))
6.                                     .add(Restrictions.lt("balance", 9200.0)))
7.                               .list();

```

Example: Equal and Not-Equal Restriction

SQL: SELECT * FROM ACCOUNT WHERE NAME <> 'somu' OR NAME = 'sekhar'

CRITERIA:

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.                               .add(Restrictions.or(
3.                                 Restrictions.ne("name", "somu"),
4.                                 Restrictions.eq("name", "sekhar")))
5.                               .list();
6.

```

OR

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.                               .add(Restrictions.or(
3.                                 Property.forName("name").ne("somu"),
4.                                 Property.forName("name").eq("sekhar")))
5.                               .list();

```

Example: Like Restriction (case-sensitive)

SQL: SELECT * FROM ACCOUNT WHERE NAME LIKE '%sekhar%' OR NAME LIKE 'cherry%'

CRITERIA:

The like operator allows wildcard searches, where the wildcard symbols are % and _, just as in SQL:

For criteria queries, wildcard searches may use either the same wildcard symbols or specify a **MatchMode**. Hibernate provides the **MatchMode** as part of the Criteria query API; we use it for writing string match expressions without string manipulation. The allowed **MatchModes** are **START**, **END**, **ANYWHERE**, and **EXACT**.

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.                               .add(Restrictions.or(
3.                                 Restrictions.like("name", "%sekhar%"),
4.                                 Restrictions.like("name", "cherry%")))
5.                               .list();

```

OR

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.                               .add(Restrictions.or(
3.                                 Restrictions.like("name", "sekhar", MatchMode.ANYWHERE),

```

```

4.             Restrictions.like("name", "cherry", MatchMode.START)))
5.         .list();

```

OR

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.     .add(Restrictions.or(
3.         Restrictions.like("name", "SEKHAR", MatchMode.ANYWHERE),
4.         Restrictions.like("name", "CHERRY", MatchMode.START)))
5.     .list();

```

OR

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.     .add(Restrictions.or(
3.         Property.forName("name").like("%sekhari%"),
4.         Property.forName("name").like("cherry%")))
5.     .list();

```

OR

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.     .add(Restrictions.or(
3.         Property.forName("name").like("sekhar", MatchMode.ANYWHERE),
4.         Property.forName("name").like("cherry", MatchMode.START)))
5.     .list();

```

Example: Like Restriction (case-insensitive)

SQL: SELECT * FROM ACCOUNT WHERE NAME LIKE '%sekhar%' AND NAME LIKE '%somu%'

CRITERIA: Re-write the above code with ilike() method instead like() method

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.     .add(Restrictions.or(
3.         Restrictions.ilike("name", "%sekhar%"),
4.         Restrictions.ilike("name", "cherry%")))
5.     .list();

```

Example: is Null, is Not Null

SQL: SELECT * FROM ACCOUNT WHERE NAME IS NULL and BALANCE IS NOT NULL

CRITERIA:

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.     .add(Restrictions.isNull("name"))
3.     .add(Restrictions.isNotNull("balance"))
4.     .list();

```

OR

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.     .add(Property.forName("name").isNull())
3.     .add(Property.forName("balance").isNotNull())
4.     .list();

```

Example: When to use Property class in general?

SQL: SELECT * FROM ACCOUNT WHERE NAME IS NOT NULL AND NAME LIKE '%sekahr%' OR NAME LIKE '%somu%'

CRITERIA: In the query there are multiple conditions on the same column.

```

1. List<Account> accounts =
2.         session.createCriteria(Account.class)
3.             .add(Restrictions.and(
4.                 Restrictions.isNotNull("name"),
5.                 Restrictions.or(
6.                     Restrictions.like("name", "sekhar", MatchMode.ANYWHERE),
7.                     Restrictions.like("name", "somu", MatchMode.ANYWHERE)))))
8.             .list();

```

OR

```

1. List<Account> accounts =
2.         session.createCriteria(Account.class)
3.             .add(Restrictions.and(
4.                 Property.forName("name").isNotNull(),
5.                 Restrictions.or(
6.                     Property.forName("name").like( "sekhar", MatchMode.ANYWHERE),
7.                     Property.forName("name").like("somu",MatchMode.ANYWHERE)))))
8.             .list();

```

OR

```

1. Property nameProperty = Property.forName("name");
2.
3. List<Account> accounts = session.createCriteria(Account.class)
4.             .add(Restrictions.and(
5.                 nameProperty.isNotNull(),
6.                 Restrictions.or(
7.                     nameProperty.like( "sekhar", MatchMode.ANYWHERE),
8.                     nameProperty.like("somu",MatchMode.ANYWHERE)))))
9.             .list();

```

NOTE: In this the last code block we are creating **nameProperty** only once, and we are reusing the same in multiple conditions, this is the exact scenario, where we need to use Property class to write conditions

Example: Between Restriction

SQL: SELECT * FROM ACCOUNT WHERE BALANCE BETWEN 4560.0 AND 8689.0

CRITERIA:

```

1. List<Account> accounts = session.createCriteria(Account.class)
2.             .add(Restrictions.between("balance", 4560.0, 8689.0))
3.             .list();

```

OR

```

4. List<Account> accounts = session.createCriteria(Account.class)
5.             .add(Property.forName("balance").between( 4560.0, 8689.0))
6.             .list();

```

Example: In Restriction

SQL: SELECT * FROM ACCOUNT WHERE ACCNO IN(5001,5009,5005,5004)

CRITERIA:

```

1. Object[] valuesArray = new Object[]{5001,5009,5005,5004};
2.
3. List<Account> accounts = session.createCriteria(Account.class)
4.                               .add(Restrictions.in("accountId", valuesArray))
5.                               .list();

```

OR

```

1. List<Integer> valuesList = new ArrayList<Integer>();
2. valuesList.add(5001);
3. valuesList.add(5009);
4. valuesList.add(5005);
5. valuesList.add(5004);
6.
7. List<Account> accounts = session.createCriteria(Account.class)
8.                               .add(Restrictions.in("accountId", valuesList))
9.                               .list();

```

Example: Property comparison

SQL: SELECT * FROM ACCOUNT WHERE ACCNO < BALANCE OR NAME LIKE '%sekhar%

CRITERIA:

```

1. Criteria criteria = session.createCriteria(Account.class);
2. Criterion criterion1 = Restrictions.ltProperty("accno", "balance");
3. Criterion criterion2 = Restrictions.like("name", "sekhar", MatchMode.ANYWHERE);
4. Criterion criterion3 = Restrictions.or(criterion1, criterion2);
5.
6. criteria.add(criterion3);
7. List<Account> accounts = criteria.list();

OR

1. Property accnoProperty = Property.forName("accno");
2. Property balanceProperty = Property.forName("balance");
3. Property nameProperty = Property.forName("name");
4.
5. Criteria criteria = session.createCriteria(Account.class);
6. Criterion criterion1 = accnoProperty.ltProperty(balanceProperty);
7. Criterion criterion2 = nameProperty.like("sekhar", MatchMode.ANYWHERE);
8. Criterion criterion3 = Restrictions.or(criterion1, criterion2);
9.
10. criteria.add(criterion3);
11. List<Account> accounts = criteria.list();

```

Example: Equal All

SQL: SELECT * FROM ACCOUNT WHERE ACCNO = 5001 AND BALANCE = 4500.0 AND NAME = 'sekhar'

CRITERIA:

```

1. Map<String, Object> map = new HashMap<String, Object>();
2. map.put("accountId", 5001);
3. map.put("name", "sekhar");
4. map.put("balance", 4500.0);
5.
6. List<Account> accounts = session.createCriteria(Account.class)
7.                               .add(Restrictions.allEq(map))
8.                               .list();

```

Example: SQL in Criteria

SQL: SELECT * FROM ACCOUNT WHERE ACCNO = 5001

CRITERIA:

```
1. List<Account> accounts = session.createCriteria(Account.class)
2.                   .add(Restrictions.sqlRestriction("{alias}.ACCNO=?",
3.                                         5001, StandardBasicTypes.INTEGER))
4.                   .list();
```

- ⇒ To write SQL syntax in criteria we use **sqlRestriction()** method of Restrictions class.
- ⇒ {alias} is because Hibernate internally generates alias name. So we should use {alias} before the column names.
- ⇒ Writing SQL syntax with Criteria is not at all advisable. Because it is database dependent.

NOTE: When we are unable to implement conditions using criteria API, then only we will go for **sqlRestrictions()**.

NOTE: In the previous version of Hibernate to represent types we used to use **org.hibernate.Hibernate** class. But this class deprecated in the latest versions, So to represent types in the latest versions we can use **org.hibernate.type.StandardBasicTypes**

Example:

SQL: SELECT * FROM ACCOUNT

WHERE

```
( ACCNO BETWEEN 5001 AND 5004 )
OR
(NAME LIKE '%sekhar%')
```

CRITERIA:

```
1. Criteria criteria = session.createCriteria(Account.class);
2. Criterion criterion1 = Restrictions.between("accountId", 5001, 5004);
3. Criterion criterion2 = Restrictions.like("name", "%sekhar%");
4. Criterion criterion3 = Restrictions.and(criterion1, criterion2);
5. criteria.add(criterion3);
6. List accounts = criteria.list();
      (OR)
1. List accounts = session.createCriteria(Account.class)
2.                   .add(Restrictions.and(
3.                           Restrictions.between("accno", 1001, 1004),
4.                           Restrictions.like("name", "%a%")))
5.                   .list();
```

Example:

SQL:SELECT * FROM ACCOUNT

WHERE

NAME IN ('sekhar', 'kesavareddy')

AND (

ACCNO IS NULL
OR ACCNO=5001
OR ACCNO=5002
OR ACCNO=5003

)

CRITERIA:

```
1. List<Account> accounts = session.createCriteria(Account.class)
```

```

2.         .add(Restrictions.in("name", new Object[]{"sekhar", "kesavareddy"}))
3.         .add(Restrictions.disjunction()
4.             .add(Restrictions.isNull("accountId"))
5.             .add(Restrictions.eq("accountId ", 5001))
6.             .add(Restrictions.eq("accountId ", 5002))
7.             .add(Restrictions.eq("accountId ", 5003)))
8.     ).list();
OR
1. Property accountId = Property.forName("accountId ");
2. List accounts = sess.createCriteria(Account.class)
3.     .add( Restrictions.disjunction()
4.         .add(accountId.isNull() )
5.         .add(accountId.eq(5001) )
6.         .add(accountId.eq(5002) )
7.         .add(accountId.eq(5003) ))
8.     .add( Property.forName("name").in( new String[] { "sekhar", "kesavareddy" } ) )
9.     .list();

```

Example:

SQL: SELECT * FROM
 (SELECT * FROM ACCOUNT
 WHERE
 NAME LIKE 'sekhar'
 ORDER BY
 NAME ASC,
 ACCNO DESC
)
 WHERE
 ROWNUM <= 5

CRITERIA:

```

1. Criteria criteria = session.createCriteria(Account.class);
2. Criterion c1 = Restrictions.like("name", "sekhar");
3. Order nameOrder = Order.asc("name");
4. Order accnoOrder = Order.desc("accountId");
5.
6. criteria.add(c1);
7. criteria.addOrder(nameOrder);
8. criteria.addOrder(accnoOrder);
9.
10. criteria.setMaxResults(5);
11.
12. List<Account> accounts = criteria.list();
OR
1. List<Account> accounts = session.createCriteria(Account.class)
2.         .add(Restrictions.like("name", "sekhar"))
3.         .addOrder(Order.asc("name"))
4.         .addOrder(Order.desc("accno"))
5.         .setMaxResults(10)
6.     .list();

```

Example: uniqueResult() method

SQL: SELECT * FROM ACCNO =5001

CRITERIA:

```
1. Account account = (Account) session.createCriteria(Account.class)
```

```

2.                               .add(Restrictions.idEq(5001))
3.                               .uniqueResult();
4.       sop(account);

```

Query By Example

- Query By **org.hibernate.criterion.Example** provides another style of searching.
- Rather than build a query by programmatically adding conditions to a Criteria object, partially populate an instance of the desired object.
 - The partially populated instance is an example.
 - Hibernate then builds the Criteria query from the example.
- To convert an object into an example, use the **create()** method in the **org.hibernate.criterion.Example** class.


```
public static Example create(Object entity);
```
- The trick with QBE is that when the query is translated into SQL, all properties of the example that are not null are used in the query.
 - To drop a property from the list, use the **excludeProperty()**method.
 - For numerical properties, use the **excludeZeros()** method.
- **Example** also has the **ignoreCase()** method, which does what it sounds like, and the **enableLike()** method, which is used for string comparisons.

Example:

SQL: SELECT * FROM ACCOUNT WHERE NAME='sekhar' AND BALANCE = 4200.0

CRITERIA:

```

1. Account account = new Account();
2. account.setName("sekhar");
3. account.setBalance(4200.0);
4.
5. Example example = Example.create(account);
6. Criteria criteria = session.createCriteria(Account.class);
7. criteria.add(example);
8. List<Account> accounts = criteria.list();

OR

1. Account account = new Account();
2. account.setName("sekhar");
3. account.setBalance(4200.0);
4.
5. Example example = Example.create(account)
   .excludeZeroes() //exclude zero valued properties
   .enableLike(MatchMode.START) //use like for string comparisons
   .excludeProperty("accountId") //exclude the property named "accountId"
   .ignoreCase(); //perform case insensitive string comparisons
10.
11. Criteria criteria = session.createCriteria(Account.class);
12. criteria.add(example);
13. List<Account> accounts = criteria.list();

```

Projections

- ⇒ Each of the above examples we selected all properties of the Entity. Instead if we want to select particular properties (columns) or to perform aggregate functions we use Projections.
- ⇒ We have done this in **HQL** also, but there we don't need any API support. As part of **HQL** query itself we can specify required properties (columns) names. But in **Criteria** we have separate API to specify required properties. i.e **Projection API**.

Q) What are the steps work with Projections in Criteria ?

Step1: Create Projection objects

```
Projection p1= Projections.property(...);
```

Step2: create ProjectionList object

```
ProjectionList pList = Projections.projectionList();
```

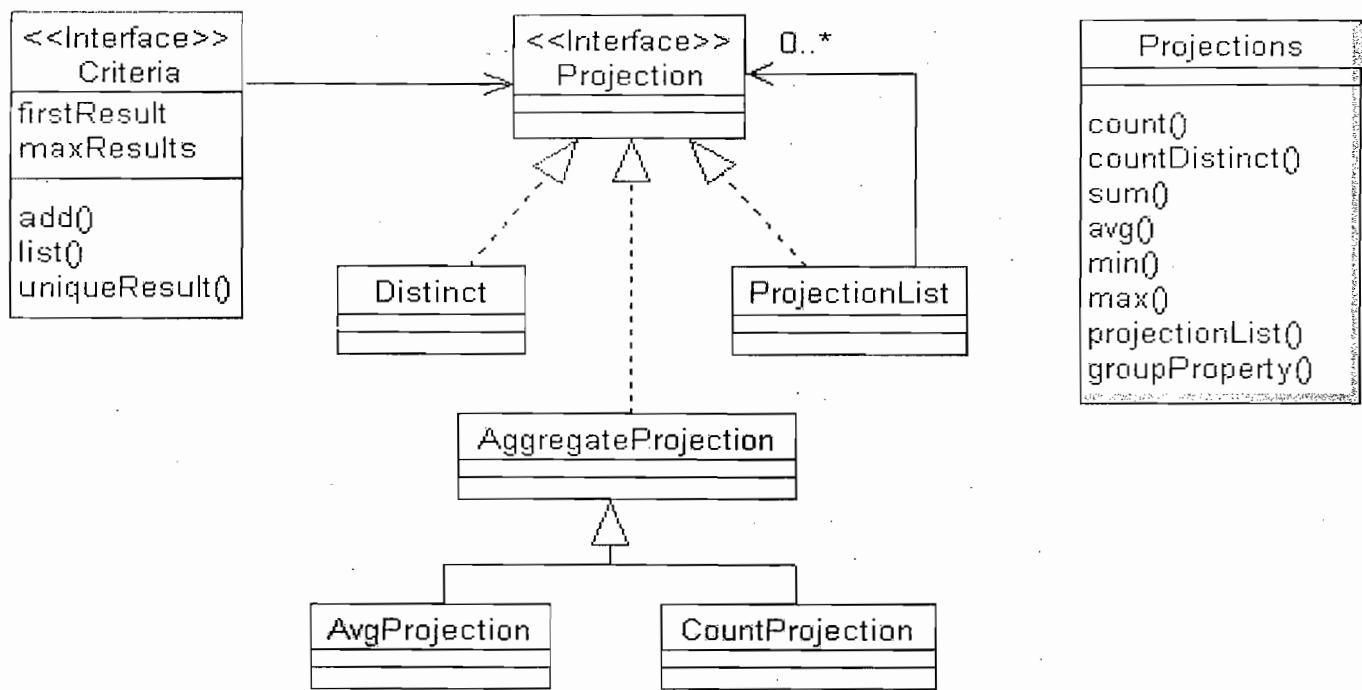
Step3: add projection objects to pList;

```
pList.add(p1);
```

Step 4: set pList to criteria object

```
criteria.setProjection(pList);
```

The following UML diagram summarizes the fundamentals of the Projections API:



- **Projection** is an interface, which represents one selected property or selected aggregate function.
- **ProjectionList** is a class, which is used to hold multiple **Projection** objects.
- **Projections** is a Util class, which contains methods to create **Projection** objects. These methods returns different **XXXPrjection** objects.
- But when we call methods on **Projections** we can assign those return types to **Projection** interface. Because all **XXXPrjection** classes directly or indirectly implements **Projection** interface.

Projection class	Method	Value/SQL function
Distinct	distinct(Projection)	DISTINCT(COLUMN)
RowCountProjection	rowCount()	COUNT(*)
CountProjection	count(property)	COUNT(COLUMN)
CountProjection	countDistinct(property)	COUNT(DISTINCT COLUMN)
AggregateProjection	max(property)	MAX(COLUMN)
AggregateProjection	min(property)	MIN(COLUMN)
AvgProjection	avg(property)	AVG(COLUMN)
AggregateProjection	sum(property)	SUM(COLUMN)
PropertyProjection	groupProperty(property)	GROUP BY <COLUMN>
IdentifierProjection	id()	<ID-COLUMN>
SQLProjection	sqlProjection(String)	SQL QUERY

Examples on Projections

Example:

SQL: SELECT NAME, BALANCE FROM ACCOUNT

CRITERIA:

```

1. Criteria criteria = session.createCriteria(Account.class);
2. ProjectionList projectionList = Projections.projectionList();
3.
4. Projection projection1 = Projections.property("name");
5. Projection projection2 = Projections.property("balance");
6.
7. projectionList.add(projection1);
8. projectionList.add(projection2);
9.
10. criteria.setProjection(projectionList);
11. List<?> list = criteria.list();

(OR)

1. List<?> list = session.createCriteria(Account.class)
           .setProjection(
               Projections.projectionList()
                   .add(Projections.property("name"))
                   .add(Projections.property("balance")))
               .list();

(OR)

1. List<?> list= session.createCriteria(Account.class)
           .setProjection(
               Projections.projectionList()
                   .add(Property.forName("name"))
                   .add(Property.forName("balance"))))
               .list();

```

NOTE: if only one selected property is there then we can set **Projection** object directly to Criteria, But if we have multiple selected properties, Then we should go for **ProjectionList**.

SQL: SELECT NAME FROM ACCOUNT

CRITERIA:

```

1. List<?> list= session.createCriteria(Account.class)
2.                     .setProjection(Projections.property("name"))
3.                     .list();
(OR)
1. List<?> list= session.createCriteria(Account.class)
2.                     .setProjection(Property.forName("name"))
3.                     .list();

```

Example:

SQL: SELECT COUNT(*) FROM ACCOUNT

CRITERIA:

```

1. List<?> list= session.createCriteria(Account.class)
2.                     .setProjection(Projections.rowCount())
3.                     .list();
4. sop("Number of rows : "+list.get(0));
(OR)
1. Long rowCount = (Long) session.createCriteria(Account.class)
2.                     .setProjection(Projections.rowCount())
3.                     .uniqueResult();
4. Sop("rowCount : " + rowCount);

```

Example:

SQL: SELECT COUNT(ACCNO) FROM ACCOUNT

CRITERIA:

```

1. List<?> list= session.createCriteria(Account.class)
2.                     .setProjection(Projections.count("accountId"))
3.                     .list();
4. sop("count : "+list.get(0));
(OR)

1. Long count =(Long) session.createCriteria(Account.class)
2.                     .setProjection(Projections.count("accountId"))
3.                     .uniqueResult();
4. sop("count : "+count);
(OR)

1. Long count =(Long) session.createCriteria(Account.class)
2.                     .setProjection(Property.forName("accountId").count())
3.                     .uniqueResult();
4. sop("count : "+count);

```

Example:

SQL: SELECT DISTINCT(NAME) FROM ACCOUNT

CRITERIA:

```

1. List<?> list= session.createCriteria(Account.class)
2.                     .setProjection(
3.                         Projections.distinct(
4.                             Projections.property("name")))
5.                     .list();
6. sop("distinct Names : "+list);

```

Example:

SQL: SELECT COUNT(DISTINCT NAME) FROM ACCOUNT

CRITERIA:

```
1. List<?> list= session.createCriteria(Account.class)
2.           .setProjection(Projections.countDistinct("name"))
3.           .list();
4. sop("count of distinct Names : "+list.get(0));
   (OR)

1. Long count =(Long) session.createCriteria(Account.class)
2.           .setProjection(Projections.countDistinct("name"))
3.           .uniqueResult();
4. sop("count of distinct Names : "+count);
```

Example:

SQL: SELECT ACCNO FROM ACCOUNT

CRITERIA:

```
1. List<?> list = session.createCriteria(Account.class)
2.           .setProjection(Projections.id())
3.           .list();
```

Example:

SQL: SELECT MAX(BALANCE), AVG(BALANCE), MIN(BLAANCE), SUM(BALANCE) FROM ACCOUNT

CRITERIA:

```
1. List<?> list = session.createCriteria(Account.class)
2.           .setProjection(
3.               Projections.projectionList()
4.               .add(Projections.max("balance"))
5.               .add(Projections.avg("balance"))
6.               .add(Projections.min("balance"))
7.               .add(Projections.sum("balance")))
8.           .list();
   OR
```

```
1. Property balanceProperty = Property.forName("balance");
2. List<?> list = session.createCriteria(Account.class)
3.           .setProjection(
4.               Projections.projectionList()
5.               .add(balanceProperty.max())
6.               .add(balanceProperty.avg())
7.               .add(balanceProperty.min()))
8.           .list();
```

Example:

SQL: SELECT COUNT(ACCNO), NAME FROM ACCOUNT GROUP BY NAME

CRITERIA:

```
1. List<?> list = session.createCriteria(Account.class)
2.           .setProjection(
3.               Projections.projectionList()
4.               .add(Projections.count("accountId"))
5.               .add(Projections.groupProperty("name")))
6.           .list();
   OR
```

```
1. List<?> list = session.createCriteria(Account.class)
2.           .setProjection(
```

```

3.             Projections.projectionList()
4.                 .add(Projections.count("accountId"))
5.                 .add(Property.forName("name").group()))
6.             .list();

```

Example:

SQL: SELECT ACCNO, NAME FROM ACCOUNT
 WHERE
 (BALANCE BETWEEN 2000 AND 4000)
 AND
 (
 NAME LIKE '%yellareddy%'
 OR
 NAME LIKE '%sekhar%'
 OR
 NAME = '%kesavareddy%'
})
 ORDER BY BALANCE DESC

CRITERIA:

```

1. List<?> list = session.createCriteria(Account.class)
2.     .setProjection(
3.         Projections.projectionList()
4.             .add(Projections.property("accountId"))
5.             .add(Projections.property("name")))
6.         .add(Restrictions.between("balance", 2000.0, 4000.0))
7.         .add(Restrictions.disjunction()
8.             .add(Restrictions.like("name", "yellareddy", MatchMode.ANYWHERE))
9.             .add(Restrictions.like("name", "sekhar", MatchMode.ANYWHERE)))
10.            .add(Restrictions.like("name", "kesavareddy", MatchMode.ANYWHERE)))
11.        .addOrder(Order.desc("balance"))
12.    .list();

```

Example: Alias Names

An alias can be assigned to a projection so that the projected value can be referred to in restrictions or orderings. Here are two different ways to do this:

```

1. List results = session.createCriteria(Account.class)
2.     .setProjection( Projections.alias( Projections.groupProperty("name"), "nm" ) )
3.     .addOrder( Order.asc("nm") )
4.     .list();
      OR
1. List results = session.createCriteria(Account.class)
2.     .setProjection( Projections.groupProperty("name").as("nm") )
3.     .addOrder( Order.asc("nm") )
4.     .list();

```

The alias() and as() methods simply wrap a projection instance in another, aliased, instance of Projection. As a shortcut, you can assign an alias when you add the projection to a projection list:

```

1. List results = session.createCriteria(Account.class)
2.     .setProjection( Projections.projectionList()
3.         .add( Projections.rowCount(), "accountCountByName" )
4.         .add( Projections.avg("balance"), "avgBalance" )
5.         .add( Projections.max("balance"), "maxBalance" )
6.         .add( Projections.groupProperty("name"), "nm" )
7.     )

```

```

8.     .addOrder( Order.desc("accountCountByName") )
9.     .addOrder( Order.desc("avgBalance") )
10.    .list();
      OR

1. List results = session.createCriteria(Account.class)
2.   .setProjection( Projections.projectionList()
3.     .add( Projections.rowCount().as("accountCountByName") )
4.     .add( Property.forName("balance").avg().as("avgBalance") )
5.     .add( Property.forName("balance").max().as("maxBalance") )
6.     .add( Property.forName("name").group().as("nm") )
7.   )
8.   .addOrder( Order.desc("accountCountByName") )
9.   .addOrder( Order.desc("avgBalance") )
10.  .list();

```

Example: Mapping criteria results to map

```

1. Map<String, Object> map = (Map<String, Object>)
2. session.createCriteria(Account.class)
3.   .setProjection(
4.     Projections.projectionList()
5.       .add(Projections.avg("balance"), "avgBalance")
6.       .add(Projections.min("balance"), "minBalance")
7.       .add(Projections.max("balance"), "maxBalance")
8.       .add(Projections.sum("balance"), "sumBalance"))
9.   . setResultTransformer(Transformers.ALIAS_TO_ENTITY_MAP)
10.  .uniqueResult();

```

Example: Mapping criteria results to map

```

1. List<Map<String, Object>> list =
2. session.createCriteria(Account.class)
3.   .setProjection(
4.     Projections.projectionList()
5.       .add(Projections.property("accountId"), "accountId")
6.       .add(Projections.property("name"), "name")
7.       .add(Projections.property("balance"), "balance"))
8.   . setResultTransformer(Transformers.ALIAS_TO_ENTITY_MAP)
9.   .list();

```

Example: Mapping criteria results to list

```

1. List<List<Object>> list = session.createCriteria(Account.class)
2.   .setProjection(
3.     Projections.projectionList()
4.       .add(Projections.property("accountId"), "accountId")
5.       .add(Projections.property("name"), "name")
6.       .add(Projections.property("balance"), "balance"))
7.   . setResultTransformer(Transformers.TO_LIST)
8.   .list();

```

Example: Mapping criteria results to bean

```

1. List<Account> list = session.createCriteria(Account.class)
2.   .setProjection(
3.     Projections.projectionList()
4.       .add(Projections.property("accountId"), "accountId")
5.       .add(Projections.property("name"), "name")
6.       .add(Projections.property("balance"), "balance"))
7.   . setResultTransformer(Transformers.aliasToBean(Account.class))
8.   .list();

```

DetachedCriteria

A **DetachedCriteria** can also be used to express a subquery. Criterion instances involving subqueries can be obtained via Subqueries or Property.

Example:

SQL: SELECT * FROM ACCOUNT WHERE BALANCE > (SELECT AVG(BALANCE) FROM ACCOUNT)

CRITERIA:

```
1. DetachedCriteria dCriteria = DetachedCriteria.forClass(Account.class);
2. Projection avgProjection = Projections.avg("balance");
3. ProjectionList pList = Projections.projectionList();
4. pList.add(avgProjection);
5. dCriteria.setProjection(pList);
6.
7. Criteria criteria = session.createCriteria(Account.class);
8. Property payProperty= Property.forName("balance");
9. Criterion criterion = payProperty.gt(dCriteria);
10.
11. criteria.add(criterion);
12. List<Account> list = criteria.list();
```

Example:

SQL: SELECT * FROM ACCOUNT WHERE > (SELECT AVG(BALANCE) FROM ACCOUNT)

CRITERIA:

```
1. DetachedCriteria requestedNames =
2. DetachedCriteria.forClass(Account.class)
3. .setProjection(Projections.property("name"))
4. .add(Restrictions.gt("accountId", 5005));
5.
6. Criteria criteria = session.createCriteria(Account.class);
7. criteria.add(Subqueries.in("name", requestedNames));
8. //criteria.add(Property.forName("name").in(requestedNames));
9. List<Account> companies = criteria.list();
```

Conclusion

- HQL is suitable for executing **Static Queries**, where as **Criteria** is suitable for executing **Dynamic Queries**
- **Criteria** used to take more time to execute than **HQL**
- Nothing is perfect, do consider your **project needs** and use it wisely.

Native SQL

- ⇒ Hibernate recommends to use **HQL** wherever possible
- ⇒ Reasons to use **Native SQL**
 - When the query is too **complex** or **impossible** to build using HQL
 - When there is no support for **new updates** of SQL in HQL. New updates mean new keywords, new style of writing queries, new features ...etc.
 - When we want call **stored procedures**
 - It also provides a clean **migration** path from a direct SQL/JDBC based application to Hibernate
- ⇒ In Native SQL we write literally SQL queries. So Native SQL is **database dependent**.
- ⇒ Hibernate3 allows you to native SQL, , for all **create, update, delete, and load** operations, including stored procedures
- ⇒ When we are writing **Native SQL** queries, we should not write pure SQL queries. The SQL queries need to be modified to include Hibernate aliases that correspond to objects or object properties.
 - Use the syntax **{objectname.*}** to refer to all properties of entity
 - Use **{objectname.property}** to refer to specific properties
 - Hibernate then uses the mapping files to translate these aliases to the proper SQL columns
- ⇒ Hibernate uses the **org.hibernate.SQLQuery** interface for native SQL
 - **SQLQuery** is a sub interface of **Query**
 - Use **createSQLQuery()** factory method on **Session** to create **SQLQuery** object.
- ⇒ The **SQLQuery** must be associated with an existing Hibernate **entity or scalar** result.

Examples:

```
1. String SQL = "SELECT * FROM ACCOUNT";
2. SQLQuery query = session.createSQLQuery(SQL);
3. List<Object[]> list = query.list();
4. for (Object[] row : list) {
5.     sop("\naccountId : "+row[0]);
6.     sop("name : "+row[1]);
7.     sop("balance : "+row[2]);
8. }
```

- ⇒ returned the results as per the order on the table
- ⇒ Here the **list** contains **Object[]** arrays.
- ⇒ One **Object[]** array per each row.
- ⇒ Each **Object[]** array size depends on the **number of columns** in the table.
- ⇒ Type of data in the **Object[]** array depends on **Dialect** class(which contains the mapping between database types and java types).

```
1.     String SQL = "SELECT * FROM ACCOUNT";
```

```

2.     SQLQuery query = session.createSQLQuery(SQL);
3.     List<Object[]> list = query.list();
4.     for (Object[] row : list) {
5.         int accountId = (Integer) row[0];
6.         String name = (String) row[1];
7.         double balance = (Double) row[2];
8.         sop("\naccountId : " + accountId);
9.         sop("name : " + name);
10.        sop("balance : " + balance);
11.    }

```

java.lang.ClassCastException: java.math.BigDecimal cannot be cast to java.lang.Integer

- ⇒ As per **Dialect** class we are using, **NUMBER** type column is mapped to **BigDecimal** of java, that's why we are getting exception.

```

1.     String SQL = "SELECT NAME, BALANCE FROM ACCOUNT";
2.     SQLQuery query = session.createSQLQuery(SQL);
3.     List<Object[]> list = query.list();
4.     for (Object[] row : list) {
5.         sop("\nname : " + row[0]);
6.         sop("balance : " + row[1]);
7.     }

```

- ⇒ Here the **list** contains **Object[]** arrays.
- ⇒ One **Object[]** array per each row.
- ⇒ Each **Object[]** array size depends on the **number of columns** we are selecting.
- ⇒ Type of data in the **Object[]** array depends on **Dialect** class(which contains the mapping between database types and java types).

```

1.     String SQL = "SELECT NAME, BALANCE FROM ACCOUNT";
2.     SQLQuery query = session.createSQLQuery(SQL);
3.     List<Object[]> list = query.list();
4.     for (Object[] row : list) {
5.         String name = (String) row[0];
6.         double balance = (Double) row[1];
7.         sop("\nname : " + name);
8.         sop("balance : " + balance);
9.     }

```

java.lang.ClassCastException: java.math.BigDecimal cannot be cast to java.lang.Double

- ⇒ As per **Dialect** class we are using, **NUMBER** type column is mapped to **BigDecimal** of java, that's why we are getting exception.

NOTE :

- Hibernate will use **ResultSetMetadata** to find the **order** and **types** of the returned values.

- How the `java.sql.Types` returned from `ResultSetMetaData` is mapped to Hibernate types is controlled by the `Dialect` class. That's why when we are trying to type cast the results to our own types we are getting `ClassCastException`.
- To avoid the overhead of using `ResultSetMetadata`, or simply to be more explicit in what is returned, one can use `addScalar()`

```

1.      String SQL = "SELECT * FROM ACCOUNT";
2.      SQLQuery query = session.createSQLQuery(SQL);
3.      query.addScalar("ACCNO", StandardBasicTypes.INTEGER);
4.      query.addScalar("NAME", StandardBasicTypes.STRING);
5.      query.addScalar("BALANCE", StandardBasicTypes.DOUBLE);
6.      List<Object[]> list = query.list();
7.      for (Object[] row : list) {
8.          int accountId = (Integer) row[0];
9.          String name = (String) row[1];
10.         double balance = (Double) row[2];
11.         sop("\naccountId : " + accountId);
12.         sop("name : " + name);
13.         sop("balance : " + balance);
14.     }

```

NOTE: In the old version of Hibernate we were using `org.hibernate.Hibernate` to specify the types. But in the latest versions we are using `org.hibernate.type.StandardBasicTypes`

```

1.      String SQL = "SELECT NAME, BALANCE FROM ACCOUNT";
2.      SQLQuery query = session.createSQLQuery(SQL);
3.      query.addScalar("NAME", StandardBasicTypes.STRING);
4.      query.addScalar("BALANCE", StandardBasicTypes.DOUBLE);
5.      List<Object[]> list = query.list();
6.      for (Object[] row : list) {
7.          String name = (String) row[0];
8.          double balance = (Double) row[1];
9.          sop("\nname : " + name);
10.         sop("balance : " + balance);
11.     }

```

NOTE: when we are calling `addScalar(alias, type)`, method we are passing column alias name, type of column. If we don't give any alias name to column, by default it will column name as alias name. If we want we can explicitly specify the column alias name.

```

1.      String SQL = "SELECT NAME AS nm, BALANCE AS bal FROM ACCOUNT";
2.      SQLQuery query = session.createSQLQuery(SQL);
3.      query.addScalar("nm", StandardBasicTypes.STRING);
4.      query.addScalar("bal", StandardBasicTypes.DOUBLE);
5.      List<Object[]> list = query.list();
6.      for (Object[] row : list) {
7.          String name = (String) row[0];

```

```

8.         double balance = (Double) row[1];
9.         sop("\nname : " + name);
10.        sop("balance : " + balance);
11.    }

```

Example: Aggregate function result mapping

```

1. String SQL = "SELECT MAX(BALANCE) AS maxBal FROM ACCOUNT ";
2. SQLQuery query = session.createSQLQuery(SQL);
3. query.addScalar("maxBal", StandardBasicTypes.DOUBLE);
4. Double maxBalance =(Double) query.uniqueResult();
5. sop("Max Balance : "+maxBalance);

```

Example: Multiple aggregate functions result mapping

```

1. String SQL = "SELECT MAX(BALANCE) AS maxBal, MIN(BALANCE) AS minBal,
                   AVG(BALANCE) AS avgBal FROM ACCOUNT ";
2. SQLQuery query = session.createSQLQuery(SQL);
3. query.addScalar("maxBal", StandardBasicTypes.DOUBLE);
4. query.addScalar("minBal", StandardBasicTypes.DOUBLE);
5. query.addScalar("avgBal", StandardBasicTypes.DOUBLE);
6. Object[] results =(Object[]) query.uniqueResult();
7. sop("Max Balance : "+results[0]);
8. sop("Min Balance : "+results[1]);
9. sop("Avg Balance : "+results[2]);

```

The following examples shows how to get entity objects from a native SQL query using **addEntity()** method

Example: Selecting all columns by using “*”

```

1. String SQL = "SELECT * FROM ACCOUNT";
2. SQLQuery query = session.createSQLQuery(SQL);
3. query.addEntity( Account.class );
4. List<Account> list = query.list();
5. for (Account account : list) {
6.     sop(account);
7. }

```

Example: Selecting all columns by specifying all the column names in the Select clause

```

1. String SQL = "SELECT ACCNO, NAME, BALANCE FROM ACCOUNT";
2. SQLQuery query = session.createSQLQuery(SQL);
3. query.addEntity( Account.class );
4. List<Account> list = query.list();
5. for (Account account : list) {
6.     sop(account);
7. }

```

Example: Select all columns by using {alias.*} syntax

```

1. String SQL = "SELECT {a.*} FROM ACCOUNT {a}";
2. SQLQuery query = session.createSQLQuery(SQL);

```

```

3. query.addEntity( "a", Account.class);
4. List<Account> list = query.list();
5. for (Account account : list) {
6.     sop(account);
7. }

```

Example: Select all columns by using COLUMN-NAME AS {alias.propertyName} syntax

```

1. String SQL = "SELECT ACCNO AS {a.accountId}, NAME AS {a.name},
2.                 BALANCE AS {a.balance} FROM ACCOUNT {a}";
3. SQLQuery query = session.createSQLQuery(SQL);
4. query.addEntity( "a", Account.class);
5. List<Account> list = query.list();
6. for (Account account : list) {
7.     sop(account);
8. }

```

Example: Select aggregate functions, assign the results to all properties of the entity

```

1. String SQL = "SELECT MAX(ACCNO) AS {a.accountId}, NAME AS {a.name},
2.                 MIN(BALANCE) AS {a.balance} FROM ACCOUNT {a} GROUP BY NAME";
3. SQLQuery query = session.createSQLQuery(SQL);
4. query.addEntity( "a", Account.class);
5. List<Account> list = query.list();
6. for (Account account : list) {
7.     sop(account);
8. }

```

NOTE: In all the above examples, we should select all columns then only the result will be assigned to Entity object. If we miss any column, Hibernate fails to assign the results to Entity objects.

⇒ Sometimes we required to select only some columns of the table, But those columns data has to get in the object format. If that is the requirement, then we can use Transformer object to map the results to our defined format.

Example: Mapping results to bean

```

1. String SQL = "SELECT ACCNO AS \"accountId\", NAME AS \"name\",
2.                 BALANCE AS \"balance\" FROM ACCOUNT ";
3. SQLQuery query = session.createSQLQuery(SQL);
4. query.addScalar("accountId", StandardBasicTypes.INTEGER);
5. query.addScalar("name", StandardBasicTypes.STRING);
6. query.addScalar("balance", StandardBasicTypes.DOUBLE);
7. query.setResultTransformer(Transformers.aliasToBean(Account.class));
8. List<Account> list = query.list();
9. for (Account account : list) {
10.     sop(account);
11. }

```

NOTE: Here alias names of the should match properties of the result bean.

Example: Mapping results to bean

```
1. String SQL = "SELECT ACCNO AS \"accountId\", BALANCE AS \"balance\"  
2. FROM ACCOUNT ";  
3. SQLQuery query = session.createSQLQuery(SQL);  
4. query.addScalar("accountId", StandardBasicTypes.INTEGER);  
5. query.addScalar("balance", StandardBasicTypes.DOUBLE);  
6. query.setResultTransformer(Transformers.aliasToBean(Account.class));  
7. List<Account> list = query.list();  
8. for (Account account : list) {  
9.     sop(account);  
10. }
```

NOTE: Here we are not selecting all the columns of the table, only selected column values will be assigned to result bean, remaining properties have default values.

Mapping results to bean which doesn't have mapping, Write the Native SQL query and assign the results to the following DTO.

```
1. package com.sekharit.hibernate.entity;  
2.  
3. public class AccountForDisplay {  
4.     private String name;  
5.     private double balance;  
6.  
7.     public AccountForDisplay() {  
8.     }  
9.  
10.    public AccountForDisplay(String name, double balance) {  
11.        super();  
12.        this.name = name;  
13.        this.balance = balance;  
14.    }  
15.  
16.    // setters & getters  
17.  
18.    @Override  
19.    public String toString() {  
20.        return "AccountForDisplay [balance=" + balance  
21.                           + ", name=" + name + "]";  
22.    }  
23.  
24. }
```

Example:

```
1. String SQL = "SELECT NAME AS \"name\", BALANCE AS \"balance\" FROM ACCOUNT ";  
2. SQLQuery query = session.createSQLQuery(SQL);  
3. query.addScalar("name", StandardBasicTypes.STRING);  
4. query.addScalar("balance", StandardBasicTypes.DOUBLE);  
5. query.setResultTransformer(Transformers.aliasToBean(AccountForDisplay.class));  
6. List<AccountForDisplay> list = query.list();  
7. for (AccountForDisplay account : list) {  
8.     sop(account);  
9. }
```

Example: Mapping results to map

```
1. String SQL = "SELECT MAX(BALANCE) maxBal, MIN(BALANCE) AS minBal,  
2.                      AVG(BALANCE) avgBal FROM ACCOUNT ";  
3. SQLQuery query = session.createSQLQuery(SQL);  
4. query.addScalar("maxBal", StandardBasicTypes.DOUBLE);  
5. query.addScalar("minBal", StandardBasicTypes.DOUBLE);  
6. query.addScalar("avgBal", StandardBasicTypes.DOUBLE);  
7. query.setResultTransformer(Transformers.ALIAS_TO_ENTITY_MAP);  
8. Map<String, Double> map = (Map<String, Double>) query.uniqueResult();  
9. sop("maxBal : "+map.get("maxBal"));  
10. sop("minBal : "+map.get("minBal"));  
11. sop("avgBal : "+map.get("avgBal"));
```

Example: Mapping results to list

```
1. String SQL = "SELECT NAME AS name, BALANCE AS balance FROM ACCOUNT ";  
2. SQLQuery query = session.createSQLQuery(SQL);  
3. query.addScalar("name", StandardBasicTypes.STRING);  
4. query.addScalar("balance", StandardBasicTypes.DOUBLE);  
5. query.setResultTransformer(Transformers.TO_LIST);  
6. List<List<Object>> list = query.list();  
7. for (List<Object> row : list) {  
8.     sop("\nName : "+ row.get(0));  
9.     sop("Balance : "+ row.get(1));  
10. }
```

NOTE : In the above Native SQL queries mostly we concentrated on mapping the results to our required format. But even we can add **conditions, positional parameters, named parameters, group by clause, having clause, order by clause, joins, inner queries...** etc. There won't be any code changes even we are adding any other clauses in the query. Because we are writing pure Native SQL queries. So our responsibility is just to map the results to our required format.

Conclusion

When the query is too complex or impossible to implement in HQL, then we can go for Native SQL.

Named Queries

- ⇒ If the query is changing or query conditions are changing frequently then instead of writing the query in java code it is advisable to move the query to **HBM** file.
- ⇒ It gives flexibility to change the queries **declaratively**.
- ⇒ We can **centralize all queries** in HBM file
- ⇒ Named queries will be **pre-compiled** at the time of HBM file loading.
- ⇒ There are two types of named Queries
 - **HQL** Named Queries
 - Native **SQL** Named Queries

HQL Named Queries

- ⇒ Writing HQL query in HBM file is called HQL Named Query.
- ⇒ For HQL writing name queries we use **<query>** tag in HBM file Or **@NamedQuery** annotation in the entity.
- ⇒ For HQL named queries we no need to map the results. Because we write HQL query on Entity name and property names.
- ⇒ In HBM file we Named query can be '**global**' or included inside **<class>** definition. If inside class definition, need to prefix with fully qualified class name when calling

Examples on HQL named queries with HBM file

Example:

SQL: SELECT * FROM ACCOUNT

HBM:

```

1. <hibernate-mapping>
2.   <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
3.     <id name="accountId" column="ACCNO"></id>
4.     <property name="name" column="NAME"></property>
5.     <property name="balance" column="BALANCE"></property>
6.   </class>
7.   <query name="findAllAccounts">FROM Account</query>
8. </hibernate-mapping>

```

Code block:

```

1. Query query = session.getNamedQuery("findAllAccounts ");
2. List<Account> accounts = query.list();
3. for (Account account : accounts) {
4.     sop(account);
5. }

```

Example:

SQL: SELECT * FROM ACCOUNT

HBM:

```
1. <hibernate-mapping>
2.   <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
3.     <id name="accountId" column="ACCNO"></id>
4.     <property name="name" column="NAME"></property>
5.     <property name="balance" column="BALANCE"></property>
6.     <query name="findAllAccounts">FROM Account</query>
7.   </class>
8. </hibernate-mapping>
```

Code block:

```
1. Query query = session.getNamedQuery("com.sekharit.hibernate.entity.Account.findAllAccounts ");
2. List<Account> accounts = query.list();
3. for (Account account : accounts) {
4.     sop(account);
5. }
```

Example:

SQL: SELECT AVG(BALANCE) FROM ACCOUNT

```
SELECT * FROM ACCOUNT WHERE NAME = 'sekhar'
SELECT * FROM ACCOUNT WHERE BALANCE = 1800
```

HBM:

```
1. <hibernate-mapping>
2.   <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
3.     <id name="accountId" column="ACCNO"></id>
4.     <property name="name" column="NAME"></property>
5.     <property name="balance" column="BALANCE"></property>
6.   </class>
7.   <query name="Account.all">FROM Account</query>
8.   <query name="Account.avgBalance">SELECT AVG(a.balance) FROM Account a</query>
9.   <query name="Account.byName">FROM Account a WHERE a.name = :name </query>
10.  <query name="Account.byBalance">FROM Account a WHERE a.balance = :balance</query>
11. </hibernate-mapping>
```

Code block:

```
1. Query query = session.getNamedQuery("Account.avgBalance");
2. double avgBalance = (Double)query.uniqueResult();
3. sop("Avg Balance : "+avgBalance);
```

Code block:

```
1. Query query = session.getNamedQuery("Account.byName");
2. query.setParameter("name", "sekhar");
3. List<Account> accounts = query.list();
4. for (Account account : accounts) {
5.     sop(account);
```

6. }

Code block:

```
1. Query query = session.getNamedQuery("Account.byBalance");
2. query.setParameter("balance", 1800.0);
3. List<Account> accounts = query.list();
4. for (Account account : accounts) {
5.     sop(account);
6. }
```

Examples on HQL named queries with annotations

Entity :

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.NamedQueries;
9. import javax.persistenceNamedQuery;
10. import javax.persistence.Table;
11.
12. import org.hibernate.annotations.GenericGenerator;
13.
14. @NamedQueries({
15.     @NamedQuery(name="Account.find.all", query= "FROM Account"),
16.     @NamedQuery(name="Account.find.allById", query= "FROM Account a WHERE a.accountId=:accountId"),
17.     @NamedQuery(name="Account.find.allByName", query= "FROM Account a WHERE a.name=:name"),
18.     @NamedQuery(name="Account.delete.byName", query= "DELETE FROM Account a WHERE
19.                                         a.name=:name"),
20.     @NamedQuery(name="Account.delete.byId", query= "DELETE FROM Account a WHERE
21.                                         a.accountId=:accountId")
22. })
23. @Entity
24. @Table(name = "ACCOUNT")
25. public class Account {
26.     @Id
27.     @GenericGenerator(name = "myGen", strategy = "increment")
28.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGen")
29.     @Column(name = "ACCNO")
30.     private int accountId;
31.     @Column(name = "NAME")
32.     private String name;
33.     @Column(name = "BALANCE")
34.     private double balance;
35.
36.     // setters & getters
```

37. }

Code block:

```
1. Query query = session.getNamedQuery("Account.find.allByName");
2. query.setParameter("name", "sekhar");
3. List<Account> accounts = query.list();
4. for (Account account : accounts) {
5.     sop(account);
6. }
```

Best Practices while using annotation based HQL named Queries

- While giving the name to the query follow some naming conventions

```
1. @NamedQueries(
2. {
3.     @NamedQuery(
4.         name="planeType.findById",
5.         query="select p from PlaneType p left join fetch p.modelPlanes where id=:id"
6.     ),
7.     @NamedQuery(
8.         name="planeType.findAll",
9.         query="select p from PlaneType p"
10.    ),
11.    @NamedQuery(
12.        name="planeType.delete",
13.        query="delete from PlaneType where id=:id"
14.    )
15. }
16. )
```

- You can put the queries into **package-info.java** class, in, say, root package of your domain objects. However, you must use Hibernate's own **@NamedQueries** and **@NamedQuery** annotations, rather than those from **javax.persistence**.

Example: package-info.java

```
@org.hibernate.annotations.NamedQueries({
    @org.hibernate.annotations.NamedQuery(
        name = "foo.findAllUsers",
        query="from Users")
})  
package com.foo.domain;
```

- Then, you have to add the package to your AnnotationConfiguration. I use Spring, so there it's a matter of setting **annotatedPackages** property: don't think that this is possible as Annotation attribute/property values must be available at compile time. Therefore, Strings cannot be

externalized to a file that needs to be read in by some sort of process. I tried to find if there was something that **package-info.java** might be able to provide, but could not find anything.

An alternative strategy for organization could be storing the queries as constants in a Class.

In your entity class:

```
@NamedQuery(name="plane.getAll", query=NamedQueries.PLANE_GET_ALL)
```

Then define a class for your query constants:

```
1. public class NamedQueries {  
2.     ...  
3.     public static final String PLANE_GET_ALL = "select p from Plane p";  
4.     ...  
5. }
```

Native SQL Named Queries

- ⇒ Writing **Native SQL** queries declaratively in HBM file is nothing but Native SQL named Queries.
- ⇒ Native SQL named queries we write using **<sql-query>** tag in HBM file, and by using **@NamedNativeQuery** annotation in the entity.
- ⇒ For Native SQL named queries we need to map the results. Because we write the Query on Table names and Column names.
- ⇒ To map the results hibernate has give tag support, annotation support.

Examples on Native SQL named queries with HBM file

Example:

SQL: SELECT * FROM ACCOUNT

HBM

```
1. <hibernate-mapping>
2.   <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
3.     <id name="accountId" column="ACCNO"></id>
4.     <property name="name" column="NAME"></property>
5.     <property name="balance" column="BALANCE"></property>
6.   </class>
7.   <sql-query name="Account.find.all">
8.     <return alias="account" class="com.sekharit.hibernate.entity.Account"></return>
9.     SELECT {account.*} from ACCOUNT {account}
10.    </sql-query>
11. </hibernate-mapping>
```

- ⇒ **<return>** tag is used to map the results to the entity

Code block:

```
1. Query query = session.getNamedQuery("Account.find.all");
2.   List<Account> accounts = query.list();
3.   for (Account account : accounts) {
4.     sop(account);
5. }
```

Example:

SQL: SELECT ACCNO, NAME, BALANCE FROM ACCOUNT

HBM

```
1. <hibernate-mapping>
2.   <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
3.     <id name="accountId" column="ACCNO"></id>
4.     <property name="name" column="NAME"></property>
5.     <property name="balance" column="BALANCE"></property>
6.   </class>
```

```
7. <sql-query name="Account.find.all">
8.   <return alias="a" class="com.sekharit.hibernate.entity.Account"></return>
9.   SELECT ACCNO AS {a.accountId}, NAME AS {a.name}, BALANCE AS {a.balance} FROM ACCOUNT {a}
10. </sql-query>
11. </hibernate-mapping>
```

Code block:

```
1. Query query = session.getNamedQuery("Account.find.all");
2.       List<Account> accounts = query.list();
3.       for (Account account : accounts) {
4.           sop(account);
5.       }
```

Example:

SQL: SELECT MAX(ACCNO), NAME, MIN(BALANCE) FROM ACCOUNT GROUP BY NAME

HBM

```
1. <hibernate-mapping>
2. <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
3.   <id name="accountId" column="ACCNO"></id>
4.   <property name="name" column="NAME"></property>
5.   <property name="balance" column="BALANCE"></property>
6. </class>
7. <sql-query name="Account.find.all">
8.   <return alias="a" class="com.sekharit.hibernate.entity.Account"></return>
9.   SELECT MAX(ACCNO) AS {a.accountId}, NAME AS {a.name}, MIN(BALANCE) AS {a.balance}
10.          FROM ACCOUNT {a} GROUP BY NAME
11. </sql-query>
12. </hibernate-mapping>
```

Code block:

```
1. Query query = session.getNamedQuery("Account.find.all");
2.       List<Account> accounts = query.list();
3.       for (Account account : accounts) {
4.           sop(account);
5.       }
```

Example: map results using column alias names

SQL: SELECT ACCNO, NAME, BALANCE FROM ACCOUNT

HBM

```
1. <hibernate-mapping>
2. <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
3.   <id name="accountId" column="ACCNO"></id>
4.   <property name="name" column="NAME"></property>
5.   <property name="balance" column="BALANCE"></property>
6. </class>
7.
```

```
8. <sql-query name="Account.find.all">
9.     <return alias="a" class="com.sekharit.hibernate.entity.Account">
10.        <return-property name="accountId" column="ano"></return-property>
11.        <return-property name="name" column="nm"></return-property>
12.        <return-property name="balance" column="bal"></return-property>
13.    </return>
14.    SELECT a.ACCNO AS ano, a.NAME AS nm, a.BALANCE AS bal FROM ACCOUNT a
15.  </sql-query>
16. </hibernate-mapping>
```

Code block:

```
1. Query query = session.getNamedQuery("Account.find.all");
2.     List<Account> accounts = query.list();
3.     for (Account account : accounts) {
4.         sop(account);
5.     }
```

Example: map results using column alias names

SQL: SELECT MAX(ACCNO), NAME, MIN(BALANCE) FROM ACCOUNT GROUP BY NAME

HBM

```
1. <hibernate-mapping>
2.   <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
3.     <id name="accountId" column="ACCNO"></id>
4.     <property name="name" column="NAME"></property>
5.     <property name="balance" column="BALANCE"></property>
6.   </class>
7.   <sql-query name="Account.find.all">
8.     <return alias="a" class="com.sekharit.hibernate.entity.Account">
9.       <return-property name="accountId" column="maxAccountId"></return-property>
10.      <return-property name="name" column="name"></return-property>
11.      <return-property name="balance" column="minBal"></return-property>
12.    </return>
13.    SELECT MAX(a.ACCNO) AS maxAccountId, a.NAME AS name, MIN(a.BALANCE) AS
14.      minBal FROM ACCOUNT {a} GROUP BY NAME
15.  </sql-query>
16. </hibernate-mapping>
```

Code block:

```
1. Query query = session.getNamedQuery("Account.find.all");
2.     List<Account> accounts = query.list();
3.     for (Account account : accounts) {
4.         sop(account);
5.     }
```

Example: map results using **<return-scalar>** tag to our required types

SQL: SELECT ACCNO, NAME, BALANCE FROM ACCOUNT

HBM

```
1. <hibernate-mapping>
2. <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
3.   <id name="accountId" column="ACCNO"></id>
4.   <property name="name" column="NAME"></property>
5.   <property name="balance" column="BALANCE"></property>
6. </class>
7. <sql-query name="Account.find.max.avg">
8.   <return-scalar column="maxBal" type="double" />
9.   <return-scalar column="avgBal" type="double" />
10.  <![CDATA[SELECT MAX(BALANCE) AS maxBal, AVG(BALANCE) AS avgBal FROM ACCOUNT]]>
11. </sql-query>
12. </hibernate-mapping>
```

Code block:

```
1. Query query = session.getNamedQuery("Account.find.max.avg");
2. List<Object[]> list = query.list();
3. for (Object[] row: list) {
4.   sop("\nMaxBalance : "+row[0]);
5.   sop("avgBalance : "+row[1]);
6. }
```

NOTE: Regarding the CDATA , it's always good practice to wrap your query text with CDATA, so that the XML parser will not prompt error for some special XML characters like '>', '<' and etc.

Examples on Native SQL named queries with HBM file

Entity:

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.EntityResult;
6. import javax.persistence.FieldResult;
7. import javax.persistence.GeneratedValue;
8. import javax.persistence.GenerationType;
9. import javax.persistence.Id;
10. import javax.persistence.NamedNativeQueries;
11. import javax.persistence.NamedNativeQuery;
12. import javax.persistence.SqlResultSetMapping;
13. import javax.persistence.SqlResultSetMappings;
14. import javax.persistence.Table;
15.
16. import org.hibernate.annotations.GenericGenerator;
17. @SqlResultSetMappings{
18.   @SqlResultSetMapping(name="myMappingOne", entities=@EntityResult(entityClass=Account.class)),
```

```
19. @SqlResultSetMapping(
20.     name="myMappingTwo",
21.     entities=@EntityResult(entityClass=Account.class,
22.                             fields={
23.                                 @FieldResult(name="accountId", column= "ano"),
24.                                 @FieldResult(name="name", column= "nm"),
25.                                 @FieldResult(name="balance", column= "bal")
26.                             }))
27. })
28.
29. @NamedNativeQueries({
30.     @NamedNativeQuery(name="Account.find.one", query= "SELECT * FROM ACCOUNT",
31.                         resultClass=Account.class),
32.     @NamedNativeQuery(name="Account.find.two", query= "SELECT ACCNO, NAME, BALANCE FROM
33.                         ACCOUNT", resultClass=Account.class),
34.     @NamedNativeQuery(name="Account.find.three", query= "SELECT * FROM ACCOUNT",
35.                         resultSetMapping="myMappingOne"),
36.     @NamedNativeQuery(name="Account.find.four", query= "SELECT ACCNO AS ano, NAME AS nm, BALANCE AS
37.                         bal FROM ACCOUNT", resultSetMapping="myMappingTwo")
38. })
39.
40. @Entity
41. @Table(name = "ACCOUNT")
42. public class Account {
43.     @Id
44.     @GenericGenerator(name = "myGen", strategy = "increment")
45.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGen")
46.     @Column(name = "ACCNO")
47.     private int accountId;
48.     @Column(name = "NAME")
49.     private String name;
50.     @Column(name = "BALANCE")
51.     private double balance;
52. }
```

Conclusion

Named queries are global access, which means the name of a query have to be unique in XML mapping files or annotations. In real environment, it's always good practice to isolate all the named queries into their own file. In addition, named queries stored in the Hibernate mapping files or annotation are more easier to maintain than queries scattered through the Java code.

Q.) What are the Relationships are there in java?

- is-a
- has-a
- use-a

is-a: If one class extends another class or one class implements another interface then that relationship is called 'is-a' relationship. For example,

Person.java

```
public class Person {  
    private String name;  
    private int age;  
  
    //setters & getters  
}
```

Employee.java

```
public class Employee extends Person {  
    private int id;  
    private double salary;  
  
    //setters & getters  
}
```

Student.java

```
public class Student extends Person {  
    private int rno;  
    private double fee;  
    private String branch;  
  
    //setters & getters  
}
```

- Here we can say Employee is-a Person, Student is-a Person.
- Person properties (name, age) are inherited into Employee and Student classes.
- In general technology, framework development time, is-a relationship is used to impose the constraints on developers for technology better usage. And is-a relationship supports dynamic polymorphism.

has-a: If one class object define in other class as a instance variable or a static variable then that relationship is called 'has-a' relationship. For example,

Car.java

```
public class Car {  
    private Gear gear;  
    private int no;  
    private String model;  
    private String color;
```

```
//setters & getters  
}
```

Gear.java

```
public class Gear {  
    private int length;  
    private String position;  
  
    //setters & getters  
}
```

- Here in this example Gear class object is defined as an instance variable in a Car class. So we can say Car 'has-a' Gear.
- In general, for the different layers integration(controller, service, DAO) we prefer to use has-a relationship. But it doesn't support dynamic polymorphism.

use-a: If one class object defined as a method parameter or a local variable in another class method then that relationship is called 'use-a' relationship. For example,

Driver.java

```
public class Driver {  
    void drive(Car c){  
        c.dodrive();  
    }  
    void drive(){  
        Bus b=new Bus();  
        b.dodrive();  
    }  
}
```

Car.java

```
public class Car {  
    void doDrive(){  
    }  
}
```

Bus.java

```
public class Bus {  
    void doDrive(){  
    }  
}
```

- Here Driver object is **using** Car & Bus objects.
- In general, for one object, if other object services are required, within one specific method, then we will go for use-a relationship.

NOTE: Using object means, calling the methods on that objects.

'has-a' relationship is again classified into follows:

1.one-to-one ----- Uni-direction
Bi-direction

2.one-to-many ----- Uni-direction
Bi-direction

3.many-to-one----- Uni-direction
Bi-direction

4.many-to-many---- Uni-direction
Bi-direction

one-to-one(unidirection):

Account.java

```
public class Account{  
    private int accno;  
    private String name;  
    private double balance;  
    private AtmCard atmcard;  
  
    //getters & setters  
}
```

AtmCard.java

```
public class AtmCard{  
    private int cno;  
    private String ctype;  
    private String issueddate;  
  
    //getters & setters  
}
```

- Here we are having the relationship from Account → AtmCard and this relationship is Uni-directional relationship.
- If we have Account object, we can access AtmCard object. But if you have AtmCard we can't access Account.

Code block:

```
Account account = // get Account object ...
```

```
AtmCard atmcard = account.getAtmCard();
```

But if we have AtmCard object, we can't get Account object.

One-to-one(Bi-directional)

Account.java

```
public class Account{  
    private int accno;  
    private String name;  
    private double balance;  
    private AccountPlus aPlus;  
  
    //getters & setters  
}
```

AccountPlus.java

```
public class AccountPlus {  
    private int acno;  
    private String acconType;  
    private String city;  
    private Account account;  
  
    //getters & setters  
}
```

- Here in this example, the relationship between Account, AccountPlus is bi-directional, so if we have Account object we can access AccountPlus object or if we have AccountPlus object we can access Account object.

Code block:

```
Account account = // get Account object ...
```

```
AccountPlus aPlus = account.getAPlus();
```

Code block:

```
AccountPlus aPlus== // get AccountPlus object ...
```

```
Account account = aPlus.getAccount();
```

One-to-Many(Uni-directional)

- If we define dependency object (independent object) as an **array** or **List** or **Set** in a dependent object then that relationship is called as One-to-Many relationship.

Book.java

```
public class Book{  
    private int bno;  
    private String btype;  
    private String author;  
    private double price;  
    private Page[] pages;  
        (or)  
    private List<Page> pages;  
        (or)  
    private Set<Page> pages;
```

```
//getters & setters  
}
```

Page.java

```
public class Page{  
    private int pno;  
    private String header;  
    private String footer;
```

```
    //getters & setters  
}
```

- Here in this example, we are defining Page[] array or List or Set defining in Book class i.e., one Book is associated with multiple Page objects. So that the relationship is called 'One-to-Many' relationship (from Book to Page).

Getting Pages from Book:

```
Book book=// get Book object...
```

```
List<Pages> pages= book.getPages();
```

How to set Pages to Book:

```
Page p1=new Page();  
Page p2=new Page();  
Page p3=newPage();
```

```
List<Page> list = new ArrayList();  
list.add(p1);  
list.add(p2);  
list.add(p3);
```

```
Book book = new Book();  
Book.setPages(list);
```

Many-to-One(Uni-directional):

Book.java

```
public class Book{  
    private int bno;  
    private String btype;  
    private String author;  
    private double price;  
  
    //getters & setters  
}
```

Page.java

```
public class Page{  
    private int pno;
```

```
private String header;  
private String footer;  
private Book book;  
  
//getters & setters  
}
```

Associating 'n' no. of Pages to one Book object:

```
Book book = new Book();  
Book.setBno(1001);  
  
Page p1=new Page();  
P1.setPno(201);  
Page p2=new Page();  
P2.setPno(202);  
P1.setBook(book);  
P2.setBook(book);
```

Getting Book object from Page:

```
Page page= // get Page object...  
Book book=page.getBook();
```

One-to-Many (or) Many-to-One (Bi-directional):-

Book.java

```
public class Book{  
    private int bno;  
    private String btype;  
    private String author;  
    private double price;  
private Set<Page> pages;  
  
    //getters & setters  
}
```

Page.java

```
public class Page{  
    private int pno;  
    private String header;  
    private String footer;  
private Book book;  
  
    //getters & setters  
}
```

Many-to-Many(Uni directional):-

Student.java

```
public class Student{  
    private int sno;  
    private String sname;  
private Course[] courses;
```

```
(or)
private List<Course> courses;
(or)
private Set<Course> courses;

// getters & setters
}

Course.java
public class Course{
    private int cno;
    private String cname;
    private String facilitator;

    // setters & getters
}
```

Code block:

```
Course c1=new Course();
c1.setCno(101);
c1.setCname("hibernate");
c1.setFacilitator("sekhar");
```

```
Course c2=new Course();
c2.setCno(102);
c2.setCname("struts");
c2.setFacilitator("sekhar");
```

```
Course c3=new Course();
c3.setCno(103);
c3.setCname("spring");
c3.setFacilitator("somu");
```

```
Set set = new HashSet();
set.add(c1);
set.add(c2);
set.add(c3);
```

```
Student s1= new Student();
s1.setSno(1001);
s1.setSname("somu");
s1.setCourses(set);
```

```
Student s2= new Student();
s2.setSno(1001);
s2.setSname("somusekhar");
s2.setCourses(set);
```

⇒ s1, s2 are two students are joining in the same courses(c1, c2, c3).

Many-to-Many(Bi-directional):-

Student.java

```
public class Student{
    private int sno;
    private String sname;
    private Course[] courses;
        (or)
    private List<Course> courses;
        (or)
    private Set<Course> courses;
    // getters & setters
}
```

Course.java

```
public class Course{
    private int cno;
    private String cname;
    private String facilitator;
    private Student[] students;
        (or)
    private List<Student> students;
        (or)
    private Set<Student> students;
    // setters & getters
}
```

DataBase Relationships

- In DataBase, there are no is-a, has-a, use-a, relationships like java.
- Primary key, foreign keys are used to form the relationships between the tables in DB.
- DataBase supports only has-a relationship of java using Primary key and foreign key.

Types of Relationships in DB:-

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

p.k → p.k	→	one-to-one
f.k(unique) → p.k	→	one-to-one
p.k → f.k	→	one-to-many
f.k → p.k	→	many-to-one
table1 → link table → table2	→	many-to-many

p.k → p.k → one-to-one

Account			AccountPlus		
ACCNO(p.k)	NAME	BALANCE	ACCNO(p.k)	CITY	ACCTYPE
1001	sekhar	5000	1001	Hyd	Salary
1002	sornu	10000	1002	B'lore	Current

f.k(unique) → p.k → one-to-one:

ACCOUNT

ACCNO(p.k)	NAME	BALANCE	A_CNO fk(unique)
1001	sekhar	5000	2001
1002	sornu	10000	2002

ATM_CARD

CNO(p.k)	CTYPE	CDATE	PASSWORD
2001	Credit	23/11/2009	*****
2002	Debit	11/23/2009	*****

p.k → f.k → one-to-many (or) f.k → p.k → many-to-one:

EMPLOYEE

EENO(p.k)	ENAME	SAL	DNO(f.k)
1001	Somu	20000	6001
1002	Sekhar	50000	6002
1003	Naresh	60000	6001

DEPARTMENT

DNO(p.k)	DNAME	LOCATION
6001	IT	Hyd
6002	HR	B'lore

table1 → link table → table2 → many-to-many

STUDENT

SNO(p.k)	SNAME
1001	Somu
1002	Naresh
1003	Sekhar

STUDENT_COURSE

SNO(f.k)	CNO(f.k)
1001	6001
1002	6002
1001	6001
1002	6002
1003	6001

COURSE

CNO(p.k)	CNAME
6001	Hibernate
6002	spring

One-to-One(pk-pk) Unidirectional

```

- oneZonePk2Pkuni
  - src
    - com.sekharit.hibernate.config
      - hibernate.cfg.xml
    - com.sekharit.hibernate.dao
      + CreateDAO.java
      + DeleteDAO.java
      + RetrieveDao.java
      + UpdateDao.java
    - com.sekharit.hibernate.entity
      + Account.java
      + AccountPlus.java
    - com.sekharit.hibernate.mapping
      + Account.hbm.xml
      + AccountPlus.hbm.xml
  - com.sekharit.hibernate.util
    + SessionUtil.java
+ JRE System Library [jre6]
+ Hibernate 4.x Library
- Referenced Libraries
  + ojdbc14.jar

```

Relation between the tables is like the following:

ACCOUNT

ACCOUNT_PLUS

SYSTEM.ACOUNT	
ACCNO	NUMBER(10)
BALANCE	FLOAT
NAME	VARCHAR2(255)

SYSTEM.ACOUNT_PLUS	
ACCNO	NUMBER(10)
LOCATION	VARCHAR2(255)
TYPE	VARCHAR2(255)

ACCOUNT	
ACCNO	NUMBER(10)
NAME	VARCHAR2(10)
BALANCE	FLOAT(126)

ACCOUNT_PLUS	
ACCNO	NUMBER(10)
LOCATION	VARCHAR2(10)
TYPE	VARCHAR2(10)

hibernate.cfg.xml

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
4. <hibernate-configuration>
5. <session-factory>
6. <property name="connection.username">system</property>

```
7. <property name="connection.url">
8.     jdbc:oracle:thin:@localhost:1521:xe
9. </property>
10. <property name="dialect">
11.     org.hibernate.dialect.Oracle9Dialect
12. </property>
13. <property name="myeclipse.connection.profile">oracle</property>
14. <property name="connection.password">tiger</property>
15. <property name="connection.driver_class">
16.     oracle.jdbc.driver.OracleDriver
17. </property>
18. <property name="show_sql">true</property>
19. <property name="hbm2ddl.auto">create</property>
20. <property name="format_sql">true</property>
21.
22. <!-- use mapping files while using configuration-->
23. <!--
24. <mapping resource="com/sekharit/hibernate/mapping/Account.hbm.xml" />
25. <mapping resource="com/sekharit/hibernate/mapping/AccountPlus.hbm.xml" />
26. -->
27.
28. <!-- use entity class while using annotations-->
29. <mapping class="com.sekharit.hibernate.entity.Account" />
30. <mapping class="com.sekharit.hibernate.entity.AccountPlus" />
31. </session-factory>
32. </hibernate-configuration>
```

Account.java

```
1. package com.sekharit.hibernate.entity;
2. package com.sekharit.hibernate.entity;
3. import javax.persistence.CascadeType;
4. import javax.persistence.Column;
5. import javax.persistence.Entity;
6. import javax.persistence.FetchType;
7. import javax.persistence.GeneratedValue;
8. import javax.persistence.GenerationType;
9. import javax.persistence.Id;
10. import javax.persistence.OneToOne;
11. import javax.persistence.PrimaryKeyJoinColumn;
12. import javax.persistence.Table;
13. import org.hibernate.annotations.GenericGenerator;
14. import org.hibernate.annotations.Parameter;
15. @Entity
16. @Table(name="ACCOUNT")
17. public class Account {
18.     @Id
19.     @GenericGenerator(
20.         name = "myGenerator",
21.         strategy = "foreign",
```

```

22.           parameters = @Parameter(name = "property", value = "plus"))
23. @GeneratedValue(strategy = GenerationType.AUTO, generator="myGenerator")
24. @Column(name="ACCNO")
25. private int accno;
26. @Column(name="NAME")
27. private String name;
28. @Column(name="BALANCE")
29. private double bal;
30.
31. @OneToOne(cascade=CascadeType.ALL,fetch=FetchType.EAGER)
32. @PrimaryKeyJoinColumn
33. private AccountPlus plus;
34.
35. //getters and setters
36. }
```

AccountPlus.java

```

1. package com.sekharit.hibernate.entity; package com.sekharit.hibernate.entity;
2. import javax.persistence.Column;
3. import javax.persistence.Entity;
4. import javax.persistence.GeneratedValue;
5. import javax.persistence.GenerationType;
6. import javax.persistence.Id;
7. import javax.persistence.Table;
8. import org.hibernate.annotations.GenericGenerator;
9. @Entity
10. @Table(name = "ACCOUNT_PLUS")
11. public class AccountPlus {
12.     @Id
13.     @GenericGenerator(name = "myGenerator", strategy = "increment")
14.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
15.     @Column(name = "ACCNO")
16.     private int accno;
17.     @Column(name = "LOCATION")
18.     private String location;
19.     @Column(name = "TYPE")
20.     private String type;
21.
22.     //getters and setters
23. }
```

Account.hbm.xml

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
6.     <id name="accno" column="ACCNO">
7.       <generator class="foreign">
```

```
8.          <param name="property">plus</param>
9.        </generator>
10.       </id>
11.       <property name="name" column="NAME" />
12.       <property name="bal" column="BALANCE" />
13.       <one-to-one name="plus" class="com.sekharit.hibernate.entity.AccountPlus"
14.                           cascade="all" />
15.     </class>
16. </hibernate-mapping>
```

AccountPlus.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.AccountPlus"
6.         table="ACCOUNT_PLUS">
7.     <id name="accno" column="ACCNO">
8.       <generator class="increment" />
9.     </id>
10.    <property name="location" column="LOCATION" />
11.    <property name="type" column="TYPE" />
12.  </class>
13. </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import org.hibernate.Transaction;
4. import com.sekharit.hibernate.entity.Account;
5. import com.sekharit.hibernate.entity.AccountPlus;
6. import com.sekharit.hibernate.util.SessionUtil;
7. public class CreateDAO {
8.   public static void main(String[] args) {
9.     Session session=SessionUtil.getSession();
10.    Transaction transaction = session.beginTransaction();
11.
12.    AccountPlus accountPlus=new AccountPlus();
13.    accountPlus.setLocation("hyd");
14.    accountPlus.setType("savings");
15.
16.    Account account=new Account();
17.    account.setName("sekhar");
18.    account.setBal(5000);
19.    account.setPlus(accountPlus);
20.
21.    session.save(account);
22.
23.    transaction.commit();
24.    SessionUtil.closeSession(session);
```

```
25. }
26. }
```

OUTPUT TABLES

ACCOUNT	ACCOUNT_PLUS				
ACCCNO	NAME	BALANCE	ACCCNO	LOCATION	TYPE
1	hyd	savings			

RetreiveDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class RetriveDAO {
6.     public static void main(String[] args) {
7.         Session session=SessionUtil.getSession();
8.         session.beginTransaction().begin();
9.
10.        Account account = (Account)session.get(Account.class, 1);
11.        System.out.println(account);
12.        sop("Account No: "+account.getAccno());
13.        sop("Name: "+account.getName());
14.        sop("Balance: "+account.getBal());
15.        sop("Location "+account.getPlus().getLocation());
16.        sop("Account Type: "+account.getPlus().getType());
17.
18.        session.getTransaction().commit();
19.        SessionUtil.closeSession(session);
20.    }
21.
22.    public static void sop(Object object){
23.        System.out.println(object);
24.    }
25. }
```

OUTPUT

Account No: 1
 Name: sekhar
 Balance: 5000.0
 Location: hyd
 Account Type: savings

UpadateDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class UpdateDAO {
6.     public static void main(String[] args) {
7.         Session session=SessionUtil.getSession();
```

As we discussed the relationships in java and database, we came to know that there is a mismatch between Database data representation and java data representation.

We can classify those mismatches as follows:

- (i) is-a relationship mismatch
- (ii) has-a relationship mismatch

has-a relationship mismatch

To overcome the has-a relationship mismatch hibernate given following tags.

<one-to-one>
<one-to-many>
<many-to-one>
<many-to-many>

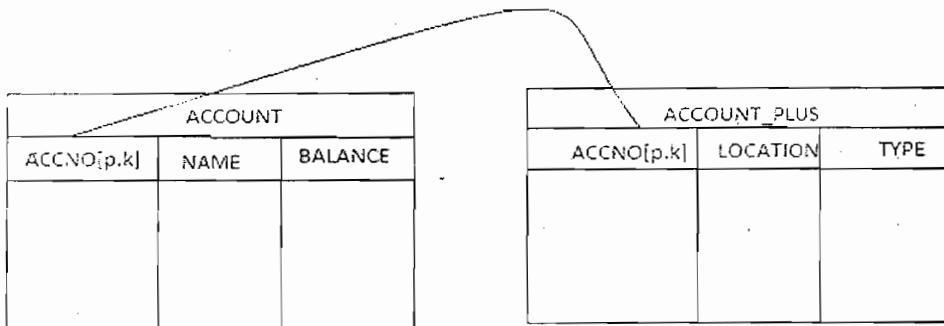
We will use hibernate given relationship tags for the following database relationship.

p.k → p.k → <one-to-one>
f.k(unique) → p.k → <many-to-one>
p.k → f.k → <one-to-many>
f.k → p.k → <many-to-one>
table1 → link table → table2 → <many-to-many>

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

Database rule is that database table should not contain more than 250 columns for performance reasons and all. In the case of Account table, it may contain more than 250 columns. In that case we write two tables for storing account information.

For example to store account information we can have two tables like ACCOUNT(ACCNO[p.k], NAME , BALANCE) and ACCOUNT_PLUS(ACCNO[p.k], LOCATION, TYPE).



In this case we should have one-to-one relationship between ACCOUNT and ACCOUNT_PLUS.

The relationship maintained between ACCOUNT.ACCNO→ ACCOUNT_PLUS.ACCNO

```

8.         session.beginTransaction().begin();
9.
10.        Account account =(Account)session.get(Account.class, 1);
11.        account.setName("somu");
12.        account.getPlus().setLocation("chennai");
13.
14.        session.getTransaction().commit();
15.        SessionUtil.closeSession(session);
16.
17.    }
18.

```

OUTPUT TABLES

ACCOUNT			ACCOUNT_PLUS		
ACCNO	NAME	BALANCE	ACCNO	LOCATION	TYPE
1	somu	5000	1	chennai	savings

DeleteDAO.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class DeleteDAO {
6.     public static void main(String[] args) {
7.         Session session=SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.        Account account =(Account)session.get(Account.class, 1);
11.        session.delete(account);
12.
13.        session.getTransaction().commit();
14.        SessionUtil.closeSession(session);
15.    }
16.

```

OUTPUT TABLE

ACCOUNT			ACCOUNT_PLUS		
ACCNO	NAME	BALANCE	ACCNO	LOCATION	TYPE
[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]	[REDACTED]

One-to-One(pk-pk) Bidirectional

```
one2onepk2pkbi
+ src
  - com.sekharit.hibernate.config
    + hibernate.cfg.xml
  - com.sekharit.hibernate.dao
    + CreateDAO1.java
    + CreateDAO2.java
    + DeleteDAO1.java
    + DeleteDAO2.java
    + RetrieveDAO1.java
    + RetrieveDAO2.java
    + UpdateDAO1.java
    + UpdateDAO2.java
  - com.sekharit.hibernate.entity
    + Account.java
    + AccountPlus.java
  - com.sekharit.hibernate.mapping
    + Account.hbm.xml
    + AccountPlus.hbm.xml
  - com.sekharit.hibernate.util
    + SessionUtil.java
+ JRE System Library [jre6]
+ Referenced Libraries
  + ojdbc14.jar
+ Hibernate 4.x Library
```

Relation between the tables is like the following:

ACCOUNT	ACCOUNT_PLUS
SYSTEM.ACOUNT ACCNO NUMBER(10) BALANCE FLOAT NAME VARCHAR2(255)	SYSTEM.ACOUNT_PLUS ACCNO NUMBER(10) LOCATION VARCHAR2(255) TYPE VARCHAR2(255)
ACCOUNT ACCNO NUMBER(10) NAME VARCHAR2(10) BALANCE FLOAT(126)	ACCOUNT_PLUS ACCNO NUMBER(10) LOCATION VARCHAR2(10) TYPE VARCHAR2(10)

Account.java

```
1. package com.sekharit.hibernate.entity;
2. import javax.persistence.CascadeType;
```

```
3. import javax.persistence.Entity;
4. import javax.persistence.FetchType;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.JoinColumn;
9. import javax.persistence.OneToOne;
10. import javax.persistence.Table;
11. import org.hibernate.annotations.GenericGenerator;
12. @Entity
13. @Table(name = "ACCOUNT")
14. public class Account {
15.     @Id
16.     @GenericGenerator(name = "myGenerator", strategy = "increment")
17.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
18.     @Column(name = "ACCNO")
19.     private int accno;
20.     @Column(name = "NAME")
21.     private String name;
22.     @Column(name = "BALANCE")
23.     private double bal;
24.     @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
25.     @PrimaryKeyJoinColumn
26.     private AccountPlus plus;
27.     // getters & setters
28. }
```

Account Plus.java

```
1. package com.sekharit.hibernate.entity;
2. import javax.persistence.CascadeType;
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.FetchType;
6. import javax.persistence.GeneratedValue;
7. import javax.persistence.GenerationType;
8. import javax.persistence.Id;
9. import javax.persistence.OneToOne;
10. import javax.persistence.PrimaryKeyJoinColumn;
11. import javax.persistence.Table;
12. import org.hibernate.annotations.GenericGenerator;
13. @Entity
14. @Table(name = "ACCOUNT_PLUS")
15. public class AccountPlus {
16.     @Id
17.     @GenericGenerator(name = "myGenerator", strategy = "increment")
18.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
19.     @Column(name = "ACCNO")
20.     private int accno;
21.     @Column(name = "LOCATION")
22.     private String location;
23.     @Column(name = "TYPE")
24.     private String type;
25.     @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
26.     @PrimaryKeyJoinColumn
27.     private Account account;
```

```
28.  
29.      //getters & setters  
30. }
```

Account.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>  
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >  
4.   <hibernate-mapping>  
5.     <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">  
6.       <id name="accno" column="ACCNO">  
7.         <generator class="increment" />  
8.       </id>  
9.       <property name="name" column="NAME" />  
10.      <property name="bal" column="BALANCE" />  
11.      <one-to-one name="plus"  
12.        class="com.sekharit.hibernate.entity.AccountPlus" cascade="all" />  
13.      </class>  
14.   </hibernate-mapping>
```

AccountPlus.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>  
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >  
4.   <hibernate-mapping>  
5.     <class name="com.sekharit.hibernate.entity.AccountPlus" table="ACCOUNT_PLUS">  
6.       <id name="accno" column="ACCNO">  
7.         <generator class="increment" />  
8.       </id>  
9.       <property name="location" column="LOCATION" />  
10.      <property name="type" column="TYPE" />  
11.      <one-to-one name="account"  
12.        class="com.sekharit.hibernate.entity.Account" cascade="all" />  
13.      </class>  
14.   </hibernate-mapping>
```

CreateDAO1.java

```
1. package com.sekharit.hibernate.dao;  
2. import org.hibernate.Session;  
3. import com.sekharit.hibernate.entity.Account;  
4. import com.sekharit.hibernate.entity.AccountPlus;  
5. import com.sekharit.hibernate.util.SessionUtil;  
6. public class CreateDAO1 {  
7.   public static void main(String[] args) {  
8.     Session session = SessionUtil.getSession();  
9.     session.beginTransaction();  
10.  
11.     AccountPlus accountPlus = new AccountPlus();  
12.     accountPlus.setLocation("hyd");  
13.     accountPlus.setType("savings");  
14.  
15.     Account account = new Account();  
16.     account.setName("sekhar");  
17.     account.setBal(5000);  
18.   }
```

```
19.         account.setPlus(accountPlus);
20.         accountPlus.setAccount(account);
21.
22.         session.save(account);
23.
24.         session.getTransaction().commit();
25.         SessionUtil.closeSession(session);
26.     }
27. }
```

OUTPUT TABLES

ACCOUNT			ACCOUNT_PLUS		
ACCNO	BALANCE	NAME	ACCNO	LOCATION	TYPE
1	5000	sekhar	1	hyd	savings

RetriveDAO1.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class RetriveDAO1 {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.         Account account = (Account) session.get(Account.class, 1);
11.         sop("Account No: "+account.getAccno());
12.         sop("Name: "+account.getName());
13.         sop("Balance: "+account.getBal());
14.         sop("Location: "+account.getPlus().getLocation());
15.         sop("Account Type: "+account.getPlus().getType());
16.
17.         session.getTransaction().commit();
18.         SessionUtil.closeSession(session);
19.
20.
21.     public static void sop(Object object) {
22.         System.out.println(object);
23.     }
24. }
```

OUTPUT

```
Account No: 1
Name: sekhar
Balance: 5000.0
Location: hyd
Account Type: savings
```

UpadateDAO1.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class UpdateDAO1 {
```

```
6. public static void main(String[] args) {
7.     Session session = SessionUtil.getSession();
8.     session.beginTransaction().begin();
9.
10.    Account account = (Account) session.get(Account.class, 1);
11.    account.setName("somasekhar");
12.    account.getPlus().setLocation("bangalore");
13.
14.    session.getTransaction().commit();
15.    SessionUtil.closeSession(session);
16.
17. }
18. }
```

OUTPUT TABLES

ACCOUNT			ACCOUNT_PLUS		
ACCCNO	BALANCE	NAME	ACCNO	LOCATION	TYPE
1	5000	somasekhar	1	bangalore	savings

DeleteDAO1.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class DeleteDAO1 {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.        Account account = (Account) session.get(Account.class, 1);
11.        session.delete(account);
12.
13.        session.getTransaction().commit();
14.        SessionUtil.closeSession(session);
15.    }
16. }
```

OUTPUT TABLES

ACCOUNT			ACCOUNT_PLUS		
ACCCNO	BALANCE	NAME	ACCCNO	LOCATION	TYPE
1	5000	somasekhar	1	bangalore	savings

CreateDAO2.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.entity.AccountPlus;
5. import com.sekharit.hibernate.util.SessionUtil;
6. public class CreateDAO2 {
7.     public static void main(String[] args) {
8.         Session session = SessionUtil.getSession();
9.         session.beginTransaction();
10.
```

```

11.         AccountPlus accountPlus = new AccountPlus();
12.         accountPlus.setLocation("hyd");
13.         accountPlus.setType("savings");
14.
15.         Account account = new Account();
16.         account.setName("sekhar");
17.         account.setBal(5000);
18.
19.         account.setPlus(accountPlus);
20.         accountPlus.setAccount(account);
21.
22.         session.save(accountPlus);
23.
24.         session.getTransaction().commit();
25.         SessionUtil.closeSession(session);
26.     }
27. }
```

OUTPUT TABLES

ACCOUNT			ACCOUNT_PLUS		
ACCNO	BALANCE	NAME	ACCNO	LOCATION	TYPE
1	5000	sekhar	1	hyd	savings

RetriveDAO2.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.util.SessionUtil;
4. import com.sekharit.hibernate.entity.AccountPlus;
5. public class RetrievetDAO2 {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.        AccountPlus plus = (AccountPlus) session.get(AccountPlus.class, 1);
11.        sop("Account No: "+plus.getAccno());
12.        sop("Location: "+plus.getLocation());
13.        sop("Account Type: "+plus.getType());
14.        sop("Name: "+plus.getAccount().getName());
15.        sop("Balance: "+plus.getAccount().getBal());
16.
17.        session.getTransaction().commit();
18.        SessionUtil.closeSession(session);
19.    }
20.
21.    public static void sop(Object object) {
22.        System.out.println(object);
23.    }
24. }
```

OUTPUT

Account No: 1
 Location: hyd
 Account Type: savings
 Name: sekhar

BALANCE: 8999.0

UpadateDAO2.java

```
1. package com.sekharit.hibernate.dao;
2. package com.sekharit.hibernate.dao;
3. import org.hibernate.Session;
4. import com.sekharit.hibernate.util.SessionUtil;
5. import com.sekharit.hibernate.entity.AccountPlus;
6. public class UpdateDAO2 {
7.     public static void main(String[] args) {
8.         Session session = SessionUtil.getSession();
9.         session.beginTransaction();
10.
11.         AccountPlus plus = (AccountPlus) session.get(AccountPlus.class, 1);
12.         plus.setLocation("anantapur");
13.         plus.getAccount().setBal(897.0);
14.
15.         session.getTransaction().commit();
16.         SessionUtil.closeSession(session);
17.
18.     }
19. }
```

OUTPUT TABLES

ACCOUNT

ACCNO	BALANCE	NAME
1	897	sekhar

ACCOUNT_PLUS

ACCNO	LOCATION	TYPE
1	anantapur	savings

DeleteDAO2.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.AccountPlus;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class DeleteDAO2 {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.         AccountPlus plus = (AccountPlus) session.get(AccountPlus.class, 1);
11.         session.delete(plus);
12.
13.         session.getTransaction().commit();
14.         SessionUtil.closeSession(session);
15.     }
16. }
```

OUTPUT TABLES

ACCOUNT

ACCNO	BALANCE	NAME	CNO

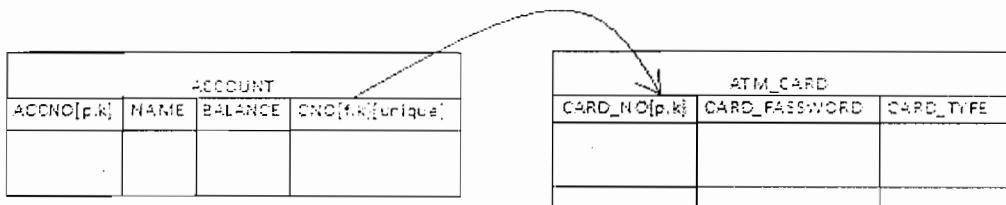
ACCOUNT_PLUS

ACCNO	LOCATION	TYPE

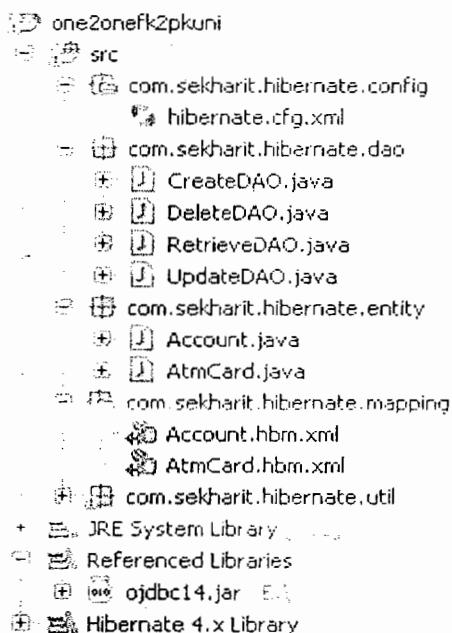


one-to-one(f.k[unique]-->p.k)

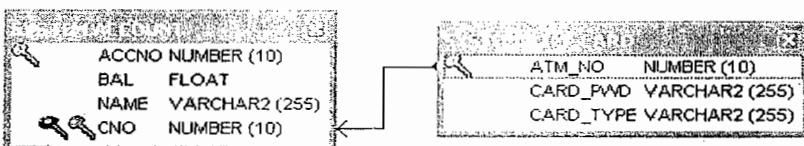
- ⇒ If we consider the relationship between ACCOUNT and ATM_CARD tables by considering the constraint, every account should contain only one ATM card.
- ⇒ Tables are
 ACCOUNT(ACCNO(p.k), NAME, BALANCE, CNO[f.k & unique])
 and
 ATM_CARD(CARD_NO[p.k], CARD_PASSWORD, CARD_TYPE)



- ⇒ In this example relationship is maintained between ACCOUNT.CNO → ATM_CARD.CARD_NO.
- ⇒ Even this relationship is between f.k → p.k it becomes one-to-one relationship. Because f.k is unique.

One-to-One(fk-pk) Unidirectional

Relation between the tables is like the following:



Account.java

```

1. package com.sekharit.hibernate.entity;
2. import javax.persistence.CascadeType;
3. import javax.persistence.Entity;
4. import javax.persistence.FetchType;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.JoinColumn;
9. import javax.persistence.OneToOne;
10. import javax.persistence.Table;
11. import org.hibernate.annotations.GenericGenerator;
12. @Entity
13. @Table(name = "ACCOUNT")
14. public class Account {
15.     @Id
16.     @GenericGenerator(name = "myGenerator", strategy = "increment")
17.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
18.     @Column(name = "ACCNO")
19.     private int accno;
20.     @Column(name = "NAME")
21.     private String name;
22.     @Column(name = "BALANCE")
23.     private double bal;
24.     @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
25.     @JoinColumn(name = "CNO", referencedColumnName = "ATM_NO", unique = true)
26.     private AtmCard atmCard;
27.
28.     // getters & setters
29. }

```

ATMCard.java

```

1. package com.sekharit.hibernate.entity;
2. import javax.persistence.CascadeType;
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.FetchType;
6. import javax.persistence.GeneratedValue;
7. import javax.persistence.GenerationType;
8. import javax.persistence.Id;
9. import javax.persistence.OneToOne;
10. import javax.persistence.PrimaryKeyJoinColumn;
11. import javax.persistence.Table;
12. import org.hibernate.annotations.GenericGenerator;
13. @Entity
14. @Table(name = "ATM_CARD")
15. public class AtmCard {
16.     @Id

```

```

17.     @GenericGenerator(name = "myGenerator", strategy = "increment")
18.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
19.     @Column(name = "ATM_NO")
20.     private int cardNo;
21.     @Column(name="CARD_PWD")
22.     private String cardPwd;
23.     @Column(name="CARD_TYPE")
24.     private String cardType;
25.
26.     //getters & setters
27. }
```

Account.hbm.xml

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
6.     <id name="accno" column="ACCNO">
7.       <generator class="foreign">
8.         <param name="property">plus</param>
9.       </generator>
10.      </id>
11.      <property name="name" column="NAME" />
12.      <property name="bal" column="BALANCE"/>
13.      <one-to-one name="plus"
14.        class="com.sekharit.hibernate.entity.AccountPlus" cascade="all" />
15.    </class>
16.  </hibernate-mapping>
```

AtmCard.hbm.xml

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.AtmCard" table="ATM_CARD">
6.     <id name="cardNo" column="CARDNO">
7.       <generator class="increment" />
8.     </id>
9.     <property name="cardPwd" column="CARDPWD"/>
10.    <property name="cardType" column="CARDTYPE"/>
11.  </class>
12. </hibernate-mapping>
```

CreateDAO.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.entity.AtmCard;
5. import com.sekharit.hibernate.util.SessionUtil;
6. public class CreateDAO {
7.   public static void main(String[] args) {
8.     Session session=SessionUtil.getSession();
9.     session.beginTransaction();
10. }
```

```

11.         AtmCard card=new AtmCard();
12.         card.setCardPwd("abc123");
13.         card.setCardType("credit");
14.
15.         Account account=new Account();
16.         account.setName("sekhar");
17.         account.setBal(1000.0);
18.         account.setAtmCard(card);
19.
20.         session.save(card);
21.
22.         session.getTransaction().commit();
23.         SessionUtil.closeSession(session);
24.     }
25. }
```

OUTPUT TABLES

ACCOUNT				ATM_CARD		
ACCNO	BAL	NAME	CNO	ATM_NO	CARD_PWD	CARD_TYPE
1	1000	sekhar	1	1	abc123	credit

RetriveDAO.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class RetriveDAO {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.         Account account = (Account) session.get(Account.class, 1);
11.
12.         sop("Account No: "+account.getAccno());
13.         sop("Name: "+account.getName());
14.         sop("balance: "+account.getBal());
15.         sop("Card No: "+account.getAtmCard().getCardNo());
16.         sop("Card Password: "+account.getAtmCard().getCardPwd());
17.         sop("Card Type: "+account.getAtmCard().getCardType());
18.
19.         session.getTransaction().commit();
20.         SessionUtil.closeSession(session);
21.     }
22.
23.     public static void sop(Object object) {
24.         System.out.println(object);
25.     }
26. }
```

OUTPUT

Account No: 1
 Name: sekhar
 Balance: 1000.0
 Card No: 1

Card Password: abc123
Card Type: credit

UpadateDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class UpdateDAO {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.        Account account = (Account) session.get(Account.class, 1);
11.        account.setName("somasekhar");
12.        account.getAtmCard().setCardPwd("pqrst");
13.
14.        session.getTransaction().commit();
15.        SessionUtil.closeSession(session);
16.
17.    }
18. }
```

OUTPUT TABLES

ACCOUNT			ATM_CARD		
ACCNO	BALANCE	NAME	CNO	ATM_NO	CARD_PWD
1	1000	somasekhar	1	1	pqrst

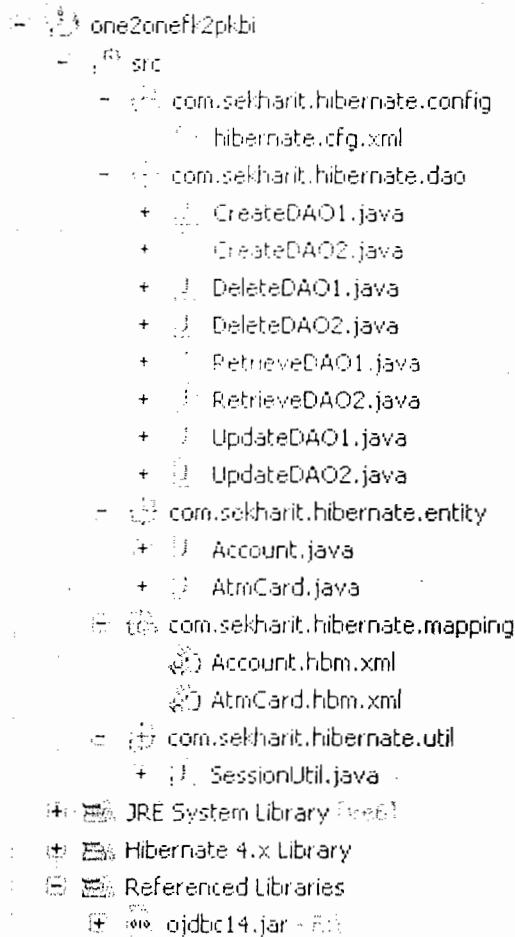
DeleteDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class DeleteDAO {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.        Account account = (Account) session.get(Account.class, 1);
11.        session.delete(account);
12.
13.        session.getTransaction().commit();
14.        SessionUtil.closeSession(session);
15.    }
16. }
```

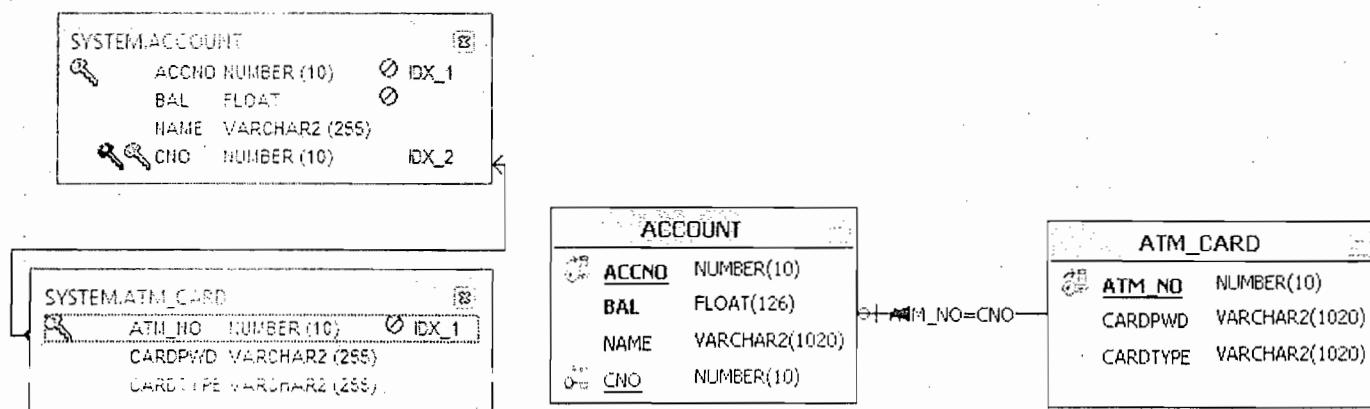
OUTPUT TABLES

ACCOUNT			ATM_CARD		
ACCNO	BALANCE	NAME	CNO	ATM_NO	CARD_PWD

One-to-One(fk-pk) Bidirectional



Relation between the tables is like the following:



Account.java

```

1. package com.sekharit.hibernate.entity;
2. import javax.persistence.CascadeType;
3. import javax.persistence.Entity;
4. import javax.persistence.FetchType;
5. import javax.persistence.GeneratedValue;

```

```
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.JoinColumn;
9. import javax.persistence.OneToOne;
10. import javax.persistence.Table;
11. import org.hibernate.annotations.GenericGenerator;
12. @Entity
13. @Table(name = "ACCOUNT")
14. public class Account {
15.     @Id
16.     @GenericGenerator(name = "myGenerator", strategy = "increment")
17.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
31.     @Column(name = "ACC_NO")
18.     private int accno;
32.     @Column(name = "NAME")
19.     private String name;
33.     @Column(name = "BAL")
20.     private double bal;
21.     @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
22.     @JoinColumn(name = "CNO", referencedColumnName = "ATM_NO", unique = true)
23.     private AtmCard atmCard;
24.
25.     // getters & setters
26. }
```

ATMCard.java

```
1. package com.sekharit.hibernate.entity;
2. import javax.persistence.CascadeType;
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.FetchType;
6. import javax.persistence.GeneratedValue;
7. import javax.persistence.GenerationType;
8. import javax.persistence.Id;
9. import javax.persistence.OneToOne;
10. import javax.persistence.PrimaryKeyJoinColumn;
11. import javax.persistence.Table;
12. import org.hibernate.annotations.GenericGenerator;
13. @Entity
14. @Table(name = "ATM_CARD")
15. public class AtmCard {
16.     @Id
17.     @GenericGenerator(name = "myGenerator", strategy = "increment")
18.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
19.     @Column(name = "ATM_NO")
20.     private int cardNo;
21.     private String cardPwd;
22.     private String cardType;
23.     @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
24.     @PrimaryKeyJoinColumn
25.     private Account account;
26.
27.     //getters & setters
28. }
```

Account.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4.   <hibernate-mapping>
5.     <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
6.       <id name="accno" column="ACCNO">
7.         <generator class="increment" />
8.       </id>
9.       <property name="name" column="NAME"/>
10.      <property name="bal" column="BALANCE"/>
11.      <many-to-one name="atmCard"
12.        class="com.sekharit.hibernate.entity.AtmCard"
13.        cascade="all" column="CNO" unique="true" fetch="join" />
14.    </class>
15.  </hibernate-mapping>
```

AtmCard.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
3.   <hibernate-mapping>
4.     <class name="com.sekharit.hibernate.entity.AtmCard" table="ATM_CARD">
5.       <id name="cardNo" column="CARDNO" length="10">
6.         <generator class="increment" />
7.       </id>
8.       <property name="cardPwd" column="CARDPWD" length="10" />
9.       <property name="cardType" column="CARDTYPE" length="10" />
10.      <one-to-one name="account"
11.        class="com.sekharit.hibernate.entity.Account"
12.        property-ref="atmCard" cascade="all">
13.      </one-to-one>
14.    </class>
15.  </hibernate-mapping>
```

CreateDAO1.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.entity.AtmCard;
5. import com.sekharit.hibernate.util.SessionUtil;
6. public class CreateDAO1 {
7.   public static void main(String[] args) {
8.     Session session=SessionUtil.getSession();
9.     session.beginTransaction();
10.
11.     AtmCard card=new AtmCard();
12.     card.setCardPwd("abcl23");
13.     card.setCardType("credit");
14.
15.     Account account=new Account();
16.     account.setName("somu");
17.     account.setBal(1000.0);
18.     account.setAtmCard(card);
19.     card.setAccount(account);
```

```

20.         session.save(card);
21.
22.         session.getTransaction().commit();
23.         SessionUtil.closeSession(session);
24.     }
25. }
```

OUTPUT TABLES

ACCOUNT				ATM_CARD		
ACCNO	BAL	NAME	CNO	ATM_NO	CARDPWD	CARDTYPE
1	1000.0	somu	1	1	1234567890	credit

RetriveDAO1.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class RetriveDAO1 {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.        Account account = (Account) session.get(Account.class, 1);
11.
12.        sop("Account No: "+account.getAccno());
13.        sop("Name: "+account.getName());
14.        sop("Balance: "+account.getBal());
15.        sop("Card No: "+account.getAtmCard().getCardNo());
16.        sop("Card Password: "+account.getAtmCard().getCardPwd());
17.        sop("Card Type: "+account.getAtmCard().getCardType());
18.
19.        session.getTransaction().commit();
20.        SessionUtil.closeSession(session);
21.    }
22.
23.    public static void sop(Object object) {
24.        System.out.println(object);
25.    }
26. }
```

OUTPUT

Account No: 1
 Name: somu
 Balance: 1000.0
 Card No: 1
 Card Password: abc123
 Card Type: credit

UpadateDAO2.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.AtmCard;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class UpdateDAO2 {
```

```

6.    public static void main(String[] args) {
7.        Session session = SessionUtil.getSession();
8.        session.beginTransaction().begin();
9.
10.       AtmCard card = (AtmCard) session.get(AtmCard.class, 1);
11.       card.setCardPwd("****");
12.       card.getAccount().setName("cherry");
13.
14.       session.getTransaction().commit();
15.       SessionUtil.closeSession(session);
16.
17.    }
18.
19. }

```

OUTPUT TABLES

ACCOUNT				ATM_CARD		
ACCNO	BAL	NAME	CNO	ATM_NO	CARDPWD	CARDTYPE
1 [REDACTED]	cherry		1 [REDACTED]	1 [REDACTED]	[REDACTED]	credit

DeleteDAO1.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class DeleteDAO1 {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction().begin();
9.
10.        Account account = (Account) session.get(Account.class, 1);
11.        session.delete(account);
12.
13.        session.getTransaction().commit();
14.        SessionUtil.closeSession(session);
15.    }
16. }

```

OUTPUT TABLES

ACCOUNT				ATM_CARD		
ACCNO	BAL	NAME	CNO	ATM_NO	CARDPWD	CARDTYPE
[REDACTED]				[REDACTED]	[REDACTED]	

CreateDAO2.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.entity.AtmCard;
5. import com.sekharit.hibernate.util.SessionUtil;
6. public class CreateDAO2 {
7.     public static void main(String[] args) {
8.         Session session = SessionUtil.getSession();
9.         session.beginTransaction().begin();
10.

```

```

11.         AtmCard card = new AtmCard();
12.         card.setCardPwd("abc123");
13.         card.setCardType("credit");
14.
15.         Account account = new Account();
16.         account.setName("somu");
17.         account.setBal(1000.0);
18.         account.setAtmCard(card);
19.         card.setAccount(account);
20.
21.         session.save(card);
22.
23.         session.getTransaction().commit();
24.         SessionUtil.closeSession(session);
25.     }
26. }
```

OUTPUT TABLES

ACCOUNT				ATM_CARD		
ACONO	BAL	NAME	CNO	ATM_NO	CARDPWD	CARDTYPE
1	1000.0	somu	1	1	abc123	credit

RetriveDAO2.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.AtmCard;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class RetriveDAO2 {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.        AtmCard card = (AtmCard) session.get(AtmCard.class, 1);
11.        sop("Card No: "+card.getCardNo());
12.        sop("Card Password: "+card.getCardPwd());
13.        sop("Card Type: "+card.getCardType());
14.        sop("Account No: "+card.getAccount().getAccno());
15.        sop("Balance: "+card.getAccount().getBal());
16.        sop("Name: "+card.getAccount().getName());
17.
18.        session.getTransaction().commit();
19.        SessionUtil.closeSession(session);
20.    }
21.
22.    public static void sop(Object object) {
23.        System.out.println(object);
24.    }
25. }
```

OUTPUT

Card No: 1
Card Password: abc123
Card Type: credit
Account No:1

Balance: 1000.0
Name: Somu

UpadateDAO1.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class UpdateDAO1 {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.        Account account = (Account) session.get(Account.class, 1);
11.        account.setName("somasekhar");
12.        account.getAtmCard().setCardPwd("pqrst");
13.
14.        session.getTransaction().commit();
15.        SessionUtil.closeSession(session);
16.
17.    }
18. }
```

OUTPUT TABLES

ACCOUNT				ATM_CARD		
ACCNO	BAL	NAME	CNO	ATM_NO	CARDPWD	CARDTYPE
1	1000.0	somasekhar	1	1	pqrst	credit

DeleteDAO2.java

```

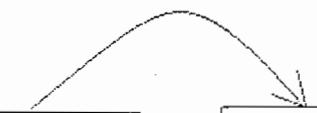
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.AtmCard;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class DeleteDAO2 {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.        AtmCard card = (AtmCard) session.get(AtmCard.class, 1);
11.        session.delete(card);
12.
13.        session.getTransaction().commit();
14.        SessionUtil.closeSession(session);
15.    }
16. }
```

OUTPUT TABLE

ACCOUNT				ATM_CARD		
ACCNO	BAL	NAME	CNO	ATM_NO	CARDPWD	CARDTYPE

Many-to-one(f.k—>p.k) & one-to-many(p.k-->f.k)

- ⇒ If we consider EMP and DEPT tables and requirement is a department can contain multiple employees.
- ⇒ Tables are
 EMPLOYEE(EID[p.k], ENAME, ESAL, EMP_DEPTNO[f.k])
 and
 DEPT(DEPTNO[p.k], DEPT_NAME, LOCATION).



EID[p.k]	ENAME	ESAL	EMP_DEPTNO[f.k]	DEPTNO[p.k]	DEPT_NAME	LOCATION
1001	stefen	12400	10	10	Account	chennai
1002	blair	24500	20	20	HR	Banglore
1003	chang	42000	10	30	Marketing	Hyderabad

- ⇒ In this example the relationship is between EMPLOYEE.EMP_DEPNO→ DEPT.DEPTNO
- ⇒ So from EMP to DEPT relationship is called as **one-to-many** relationship.
- ⇒ If we consider from DEPT to EMP then this relationship is called **many-to-one** relationship.
- ⇒ The relationship is to be maintained based on the screens developed in the projects. And based on the way we insert the records into Database tables.

Consider the following Screen

Employee Registration Screen

EID :	
Name :	
salary :	
Designation :	
Department :	<input type="checkbox"/> Accounts <input type="checkbox"/> Sales <input type="checkbox"/> IT <input type="checkbox"/> Marketing
<input type="button" value="Submit"/>	

- ⇒ In the above screen we can submit multiple employees information by selecting same department.
- ⇒ Many-to-one doesn't mean that, At a time many Employees are inserted with single Department. At a time only one Employee will be inserted by associating with one Department. After inserting multiple Employees by associating same department, in database we can find that multiple employees are pointing single department.
- ⇒ And generally DEPT table designed and given the data before itself. Because DEPT table doesn't change frequently. These type of tables are called lookup tables or constant tables.
- ⇒ And this table data has to be displayed in the drop down box of our screen.
- ⇒ Consider the following screen

Department and Employee Registration Screen

DEPTNO :	<input type="text"/>	DNAME :	<input type="text"/>	LOCATION :	<input type="text"/>
<input type="button" value="Add More"/>					
EID :	<input type="text"/>				
Name :	<input type="text"/>				
salary :	<input type="text"/>				
Designation :	<input type="text"/>				
EID :	<input type="text"/>				
Name :	<input type="text"/>				
salary :	<input type="text"/>				
Designation :	<input type="text"/>				
<input type="button" value="Submit"/>					

- ⇒ So by considering the different screens, we can say, the relationship need to be maintained between the entities completely depends on screens design and the way we insert the records into the database tables.

One-to-Many relationship can be developed in four ways.

- Set
- List
- Bag
- Array

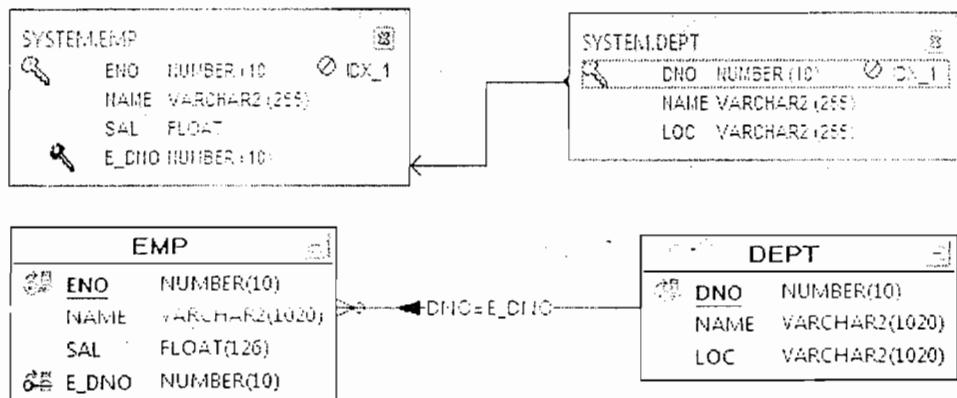
one-to-many (set) unidirectional(Department to Employee)

```

.
.
.
└ com.sekharit
   └ one2many_setuni
      └ src
         └ com.sekharit.hibernate.config
            └ hibernate.cfg.xml
         └ com.sekharit.hibernate.dao
            └ CreateDAO.java
            └ DeleteDAO.java
            └ RetrieveDAO.java
            └ UpdateDAO.java
         └ com.sekharit.hibernate.entity
            └ Department.java
            └ Employee.java
         └ com.sekharit.hibernate.mapping
            └ Department.hbm.xml
            └ Employee.hbm.xml
         └ com.sekharit.hibernate.util
            └ SessionUtil.java
      └ JRE System Library [JRE]
      └ Hibernate 4.x Libraries
      └ Referenced Libraries

```

Relation between the tables is like the following:

Department.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import java.util.List;
4. import javax.persistence.CascadeType;
5. import javax.persistence.Column;
6. import javax.persistence.Entity;
7. import javax.persistence.FetchType;
8. import javax.persistence.GeneratedValue;
9. import javax.persistence.GenerationType;
10. import javax.persistence.Id;
11. import javax.persistence.JoinColumn;
12. import javax.persistence.OneToMany;
13. import javax.persistence.Table;
14.

```

```
15. import org.hibernate.annotations.GenericGenerator;
16. import org.hibernate.annotations.IndexColumn;
17.
18. @Entity
19. @Table(name = "DEPT")
20. public class Department {
21.     @Id
22.         @GenericGenerator(name = "myGenerator", strategy = "increment")
23.         @GeneratedValue(strategy=GenerationType.AUTO, generator="myGenerator")
24.         @Column(name = "DNO")
25.         private int dno;
26.         @Column(name = "NAME")
27.         private String name;
28.         @Column(name = "LOC")
29.         private String location;
30.         @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
31.             @JoinColumn(name = "E_DNO", referencedColumnName = "DNO")
32.             private Set<Employee> employeeSet;
33.
34.     //getters and setters
35. }
```

Employee.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.Table;
9.
10. import org.hibernate.annotations.GenericGenerator;
11.
12. @Entity
13. @Table(name = "EMP")
14. public class Employee {
15.     @Id
16.         @GenericGenerator(name = "myGenerator", strategy = "increment")
17.         @GeneratedValue(strategy=GenerationType.AUTO, generator="myGenerator")
18.         @Column(name = "ENO")
19.         private int eno;
20.         @Column(name = "NAME")
21.         private String name;
22.         @Column(name = "SAL")
23.         private double salary;
24.
25.
26.     //getters and setters
27. }
```

Department.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
```

```
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Department" table="DEPT">
6.     <id name="dno" column="DNO">
7.       <generator class="increment" />
8.     </id>
9.     <property name="name" column="NAME" />
10.    <property name="location" column="LOC" />
11.    <set name="employeeSet" cascade="all">
12.      <key column="E_DNO" />
13.      <one-to-many class="com.sekharit.hibernate.entity.Employee" />
14.    </set>
15.  </class>
16. </hibernate-mapping>
```

Employee.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3.   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Employee" table="EMP">
6.     <id column="ENO" name="eno">
7.       <generator class="increment" />
8.     </id>
9.     <property column="NAME" name="name" />
10.    <property column="SAL" name="salary" />
11.  </class>
12. </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import java.util.HashSet;
4. import java.util.Set;
5.
6. import org.hibernate.Session;
7.
8. import com.sekharit.hibernate.entity.Department;
9. import com.sekharit.hibernate.entity.Employee;
10. import com.sekharit.hibernate.util.SessionUtil;
11.
12. public class CreateDAO {
13.     public static void main(String[] args) {
14.         Session session = SessionUtil.getSession();
15.         session.beginTransaction();
16.
17.         Employee employee1 = new Employee();
18.         employee1.setName("kesavareddy");
19.         employee1.setSalary(4600);
20.
21.         Employee employee2 = new Employee();
22.         employee2.setName("yellareddy");
23.         employee2.setSalary(5800);
24.     }
25. }
```

```

25.         Employee employee3 = new Employee();
26.         employee3.setName("cherry");
27.         employee3.setSalary(5900);
28.
29.         Set<Employee> empSet = new HashSet<Employee>();
30.         empSet.add(employee1);
31.         empSet.add(employee2);
32.         empSet.add(employee3);
33.
34.         Department department = new Department();
35.         department.setName("Agriculture");
36.         department.setLocation("Gunipalli");
37.         department.setEmployeeSet(empSet);
38.
39.         session.save(department);
40.
41.         session.getTransaction().commit();
42.         SessionUtil.closeSession(session);
43.     }
44. }
```

OUTPUT TABLES:

EMP				DEPT		
E_NO	NAME	SAL	E_DNO	DNO	NAME	LOC
1	kesavareddy	4600	1			
2	yellareddy	5800	1			
3	cherry	5900	1	Agriculture	Gunipalli	

RetriveDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import java.util.Set;
4.
5. import org.hibernate.Session;
6.
7. import com.sekharit.hibernate.entity.Department;
8. import com.sekharit.hibernate.entity.Employee;
9. import com.sekharit.hibernate.util.SessionUtil;
10.
11. public class RetriveDAO {
12.     public static void main(String[] args) {
13.         Session session = SessionUtil.getSession();
14.         session.beginTransaction();
15.
16.         Department department = (Department) session.get(Department.class, 1);
17.         Set<Employee> empSet = department.getEmployeeSet();
18.         sop("Department details are...");
19.         sop("Dno : " + department.getDno());
20.         sop("Name : " + department.getName());
21.         sop("Location : " + department.getLocation());
22. }
```

```
23.         sop("\nEmployee details are... ");
24.         for (Employee employee : empSet) {
25.             sop("\nEno : " + employee.getEno());
26.             sop("Name : " + employee.getName());
27.             sop("Salary : " + employee.getSalary());
28.         }
29.
30.         session.getTransaction().commit();
31.         SessionUtil.closeSession(session);
32.     }
33.
34.     public static void sop(Object object) {
35.         System.out.println(object);
36.     }
37. }
```

OUTPUT:

Department details are...

Dno : 1
Name : Agriculture
Location : Gunipalli

Eno : 1
Name : kesavareddy
Salary : 4600.0

Eno : 2
Name : yellareddy
Salary : 5800.0

Eno : 3
Name : cherry
Salary : 5900.0

UpadateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import java.util.Iterator;
4. import java.util.Set;
5.
6. import org.hibernate.Session;
7.
8. import com.sekharit.hibernate.entity.Department;
9. import com.sekharit.hibernate.entity.Employee;
10. import com.sekharit.hibernate.util.SessionUtil;
11.
12. public class UpdateDAO {
13.     public static void main(String[] args) {
14.         Session session = SessionUtil.getSession();
15.         session.beginTransaction();
16.
17.         Department department = (Department) session.get(Department.class, 1);
18.         department.setName("IT");
19.         Set<Employee> empSet = department.getEmployeeSet();
20.         Iterator<Employee> iterator = empSet.iterator();
21.         while (iterator.hasNext()) {
```

```

22.             Employee employee = iterator.next();
23.             if (employee.getSalary() > 5200) {
24.                 employee.setSalary(employee.getSalary() + 1000);
25.             }
26.         }
27.
28.         session.getTransaction().commit();
29.         SessionUtil.closeSession(session);
30.     }
31. }
```

OUTPUT TABLES:

EMP				DEPT		
ENO	NAME	SAL	E_DNO	DNO	NAME	LOC
1	kesavareddy	4600	1	1	IT	Gunipalli
2	yellareddy	6800	1			
3	cherry	6900	1			

DeleteDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.util.SessionUtil;
7.
8. public class DeleteDAO {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Department department = (Department) session.get(Department.class, 1);
14.         session.delete(department);
15.
16.         session.getTransaction().commit();
17.         SessionUtil.closeSession(session);
18.     }
19. }
```

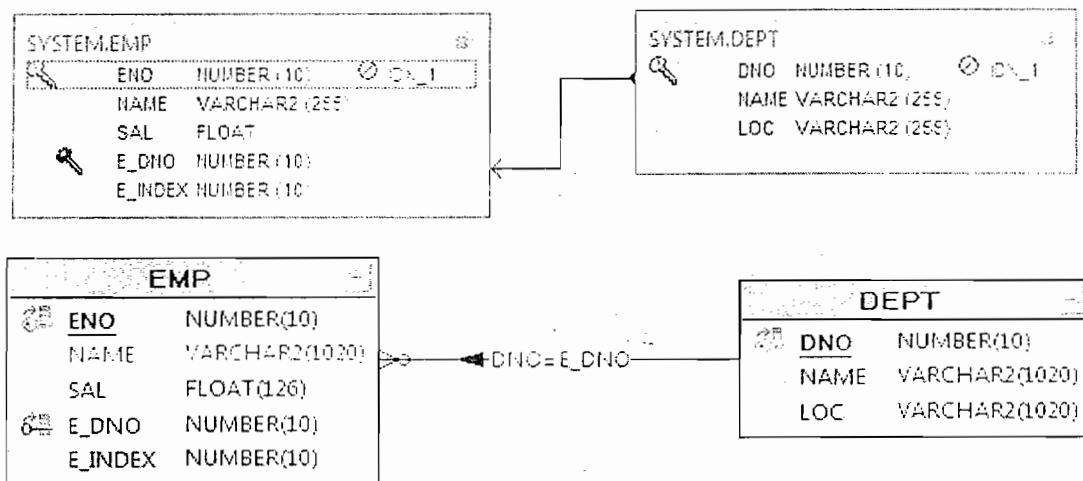
OUTPUT TABLES:

EMP				DEPT		
ENO	NAME	SAL	E_DNO	DNO	NAME	LOC
1	kesavareddy	4600	1	1	IT	Gunipalli

One-to-many (list) unidirectional(Department to Employee)

A project tree diagram showing the structure of the 'one2manylistuni' project. The root node is 'one2manylistuni'. It contains a 'src' folder, which in turn contains several packages: 'com.sekharit.hibernate.config', 'com.sekharit.hibernate.dao', 'com.sekharit.hibernate.entity', 'com.sekharit.hibernate.mapping', and 'com.sekharit.hibernate.util'. Each package has its corresponding Java files listed below it. Additionally, there are three external library nodes: 'JRE System Library', 'Hibernate 4.x Libraries', and 'Referenced Libraries'.

Relation between the tables is like the following:



Department.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.CascadeType;
4. import javax.persistence.Column;
5. import javax.persistence.Entity;
6. import javax.persistence.FetchType;
7. import javax.persistence.GeneratedValue;
8. import javax.persistence.GenerationType;
9. import javax.persistence.Id;
10. import javax.persistence.JoinColumn;
11. import javax.persistence.OneToMany;
12. import javax.persistence.Table;
13.
14. import org.hibernate.annotations.GenericGenerator;
15. import org.hibernate.annotations.IndexColumn;
```

```

16.
17. @Entity
18. @Table(name = "DEPT")
19. public class Department {
20.     @Id
21.     @GenericGenerator(name = "myGenerator", strategy = "increment")
22.     @GeneratedValue(strategy=GenerationType.AUTO, generator="myGenerator")
23.     @Column(name = "DNO",
24.     private int dno;
25.     @Column(name = "NAME")
26.     private String name;
27.     @Column(name = "LOC")
28.     private String location;
29.     @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
30.     @JoinColumn(name = "E_DNO", referencedColumnName = "DNO")
31.     @IndexColumn(name = "E_INDEX")
32.     private Employee[] empArray;
33.
34.     //getters and setters
35. }

```

Employee.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.Table;
9.
10. import org.hibernate.annotations.GenericGenerator;
11.
12. @Entity
13. @Table(name = "EMP")
14. public class Employee {
15.     @Id
16.     @GenericGenerator(name = "myGenerator", strategy = "increment")
17.     @GeneratedValue(strategy=GenerationType.AUTO, generator="myGenerator")
18.     @Column(name = "ENO",
19.     private int eno;
20.     @Column(name = "NAME")
21.     private String name;
22.     @Column(name = "SAL")
23.     private double salary;
24.
25.
26.     //getters and setters
27. }

```

Department.hbm.xml

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>

```

```
5. <class name="com.sekharit.hibernate.entity.Department" table="DEPT">
6.   <id name="dno" column="DNO">
7.     <generator class="increment" />
8.   </id>
9.   <property name="name" column="NAME" />
10.  <property name="location" column="LOC" />
11.  <array name="empArray" cascade="all">
12.    <key column="E_DNO" />
13.    <index column="E_INDEX"></index>
14.    <one-to-many class="com.sekharit.hibernate.entity.Employee" />
15.  </array>
16. </class>
17. </hibernate-mapping>
```

Employee.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3.   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Employee" table="EMP">
6.     <id column="ENO" name="eno">
7.       <generator class="increment" />
8.     </id>
9.     <property column="NAME" name="name" />
10.    <property column="SAL" name="salary" />
11.  </class>
12. </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. import org.hibernate.Session;
7.
8. import com.sekharit.hibernate.entity.Department;
9. import com.sekharit.hibernate.entity.Employee;
10. import com.sekharit.hibernate.util.SessionUtil;
11.
12. public class CreateDAO {
13.   public static void main(String[] args) {
14.     Session session = SessionUtil.getSession();
15.     session.beginTransaction();
16.
17.     Employee employee1 = new Employee();
18.     employee1.setName("kesavareddy");
19.     employee1.setSalary(4600);
20.
21.     Employee employee2 = new Employee();
22.     employee2.setName("yellareddy");
23.     employee2.setSalary(5800);
24. }
```

```

25.         Employee employee3 = new Employee();
26.         employee3.setName("cherry");
27.         employee3.setSalary(5910);
28.
29.         List<Employee> empList = new ArrayList<Employee>();
30.         empList.add(employee1);
31.         empList.add(employee2);
32.         empList.add(employee3);
33.
34.         Department department = new Department();
35.         department.setName("Agriculture");
36.         department.setLocation("Gunipalli");
37.         department.setEmployeeList(empList);
38.
39.         session.save(department);
40.
41.         session.getTransaction().commit();
42.         SessionUtil.closeSession(session);
43.     }
44. }
```

OUTPUT TABLES:

EMP				
E_NO	NAME	SAL	E_DNO	E_INDEX
1 somu		25000	1	0
2 sekhar		35000	1	1
3 somasekhar		60000	1	2

DEPT		
DNO	NAME	LOC
1 Accounts		Anantapur

RetriveDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import java.util.List;
4.
5. import org.hibernate.Session;
6.
7. import com.sekharit.hibernate.entity.Department;
8. import com.sekharit.hibernate.entity.Employee;
9. import com.sekharit.hibernate.util.SessionUtil;
10.
11. public class RetriveDAO {
12.     public static void main(String[] args) {
13.         Session session = SessionUtil.getSession();
14.         session.beginTransaction();
15.
16.         Department department = (Department) session.get(Department.class, 1);
17.         List<Employee> empList = department.getEmployeeList();
18.         sop("Department details are...");
19.         sop("Dno : " + department.getDno());
20.         sop("Name : " + department.getName());
21.         sop("Location : " + department.getLocation());
```

```
22.           sop("\nEmployee details are... ");
23.           for (Employee employee : empList) {
24.               sop("\nEno : " + employee.getEno());
25.               sop("Name : " + employee.getName());
26.               sop("Salary : " + employee.getSalary());
27.           }
28.
29.
30.           session.getTransaction().commit();
31.           SessionUtil.closeSession(session);
32.       }
33.
34.   public static void sop(Object object) {
35.       System.out.println(object);
36.   }
37. }
```

OUTPUT:

Department details are...

Dno : 1
Name : Accounts
Location : Anantapur

Employee details are...

Eno : 1
Name : somu
Salary : 25000.0

Eno : 2
Name : sekhar
Salary : 35000.0

Eno : 3
Name : somasekhar
Salary : 60000.0

UpadateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import java.util.Iterator;
4. import java.util.List;
5.
6. import org.hibernate.Session;
7.
8. import com.sekharit.hibernate.entity.Department;
9. import com.sekharit.hibernate.entity.Employee;
10. import com.sekharit.hibernate.util.SessionUtil;
11.
12. public class UpdateDAO {
13.     public static void main(String[] args) {
14.         Session session = SessionUtil.getSession();
15.         session.beginTransaction();
16.
17.         Department department = (Department) session.get(Department.class, 1);
18.         department.setName("IT");
```

```

19.         List<Employee> empList = department.getEmployeeList();
20.
21.         Iterator<Employee> iterator = empList.iterator();
22.         while (iterator.hasNext()) {
23.             Employee employee = iterator.next();
24.             if (employee.getSalary() > 5200) {
25.                 employee.setSalary(employee.getSalary() + 1000);
26.             }
27.         }
28.
29.         session.getTransaction().commit();
30.         SessionUtil.closeSession(session);
31.     }
32. }
```

OUTPUT TABLES:

EMP

ENO	NAME	SAL	E_DNO	E_INDEX
1	somu	25000	1	0
2	sekharreddy	35000	1	1
3	sekharreddy	60000	1	2

DEPT

DNO	NAME	LOC
1	Accounts	Hyderabad

DeleteDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.util.SessionUtil;
7.
8. public class DeleteDAO {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Department department = (Department) session.get(Department.class, 1);
14.         session.delete(department);
15.
16.         session.getTransaction().commit();
17.         SessionUtil.closeSession(session);
18.     }
19. }
```

OUTPUT TABLES:

EMP

ENO	NAME	SAL	E_DNO	E_INDEX
1	somu	25000	1	0

DEPT

DNO	NAME	LOC
1	Accounts	Hyderabad

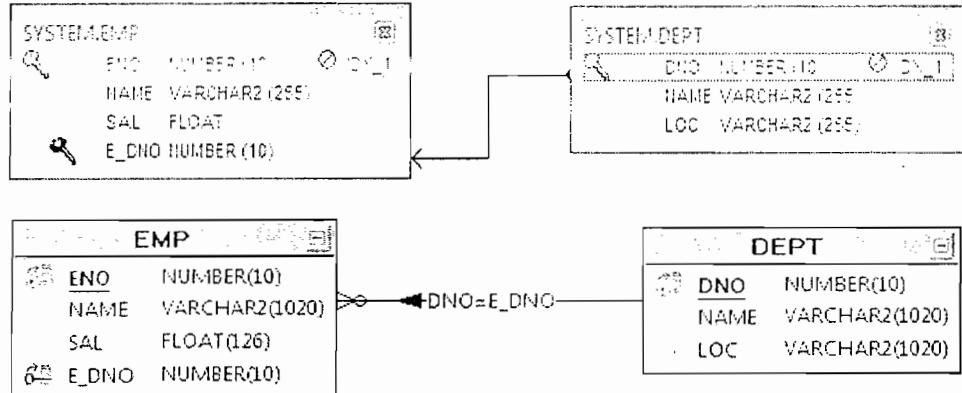
One-to-many(bag) unidirectional(Department to Employee)

```

1. one2man/baguni
  2. src
    - com.sekharit.hibernate.config
      - hibernate.cfg.xml
    - com.sekharit.hibernate.dao
      - CreateDAO.java
      - DeleteDAO.java
      - RetrieveDAO.java
      - UpdateDAO.java
    - com.sekharit.hibernate.entity
      - Department.java
      - Employee.java
    - com.sekharit.hibernate.mapping
      - Department.hbm.xml
      - Employee.hbm.xml
  - com.sekharit.hibernate.util
    - SessionUtil.java
  - JRE System Library [JRE]
  - Hibernate 4.x Libraries
  - Referenced Libraries

```

Relation between the tables is like the following:

Department.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import java.util.List;
4. import javax.persistence.CascadeType;
5. import javax.persistence.Column;
6. import javax.persistence.Entity;
7. import javax.persistence.FetchType;
8. import javax.persistence.GeneratedValue;
9. import javax.persistence.GenerationType;
10. import javax.persistence.Id;
11. import javax.persistence.JoinColumn;
12. import javax.persistence.OneToMany;
13. import javax.persistence.Table;

```

```
14.  
15. import org.hibernate.annotations.GenericGenerator;  
16. import org.hibernate.annotations.IndexColumn;  
17.  
18. @Entity  
19. @Table(name = "DEPT")  
20. public class Department {  
21.     @Id  
22.     @GenericGenerator(name = "myGenerator", strategy = "increment")  
23.     @GeneratedValue(strategy=GenerationType.AUTO, generator="myGenerator")  
24.     @Column(name = "DNO")  
25.     private int dno;  
26.     @Column(name = "NAME",  
27.     private String name;  
28.     @Column(name = "LOC")  
29.     private String location;  
30.     @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)  
31.     @JoinColumn(name = "E_DNO", referencedColumnName = "DNO")  
32.     private List<Employee> employeeList;  
33.  
34.     //getters and setters  
35. }
```

Employee.java

```
1. package com.sekharit.hibernate.entity;  
2.  
3. import javax.persistence.Column;  
4. import javax.persistence.Entity;  
5. import javax.persistence.GeneratedValue;  
6. import javax.persistence.GenerationType;  
7. import javax.persistence.Id;  
8. import javax.persistence.Table;  
9.  
10. import org.hibernate.annotations.GenericGenerator;  
11.  
12. @Entity  
13. @Table(name = "EMP")  
14. public class Employee {  
15.     @Id  
16.     @GenericGenerator(name = "myGenerator", strategy = "increment")  
17.     @GeneratedValue(strategy=GenerationType.AUTO, generator="myGenerator")  
18.     @Column(name = "ENO")  
19.     private int eno;  
20.     @Column(name = "NAME")  
21.     private String name;  
22.     @Column(name = "SAL")  
23.     private double salary;  
24.  
25.  
26.     //getters and setters  
27. }
```

Department.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>  
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
```

```
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Department" table="DEPT">
6.     <id name="dno" column="DNO">
7.       <generator class="increment" />
8.     </id>
9.     <property name="name" column="NAME" />
10.    <property name="location" column="LOC" />
11.    <bag name="employeeList" cascade="all">
12.      <key column="E_DNO" />
13.      <one-to-many class="com.sekharit.hibernate.entity.Employee" />
14.    </bag>
15.  </class>
16. </hibernate-mapping>
```

Employee.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3.   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Employee" table="EMP">
6.     <id column="ENO" name="eno">
7.       <generator class="increment" />
8.     </id>
9.     <property column="NAME" name="name" />
10.    <property column="SAL" name="salary" />
11.  </class>
12. </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. import org.hibernate.Session;
7.
8. import com.sekharit.hibernate.entity.Department;
9. import com.sekharit.hibernate.entity.Employee;
10. import com.sekharit.hibernate.util.SessionUtil;
11.
12. public class CreateDAO {
13.     public static void main(String[] args) {
14.         Session session = SessionUtil.getSession();
15.         session.beginTransaction();
16.
17.         Employee employee1 = new Employee();
18.         employee1.setName("kesavareddy");
19.         employee1.setSalary(4600);
20.
21.         Employee employee2 = new Employee();
22.         employee2.setName("yellareddy");
23.         employee2.setSalary(5800);
```

```

24.
25.     Employee employee3 = new Employee();
26.     employee3.setName("cherry");
27.     employee3.setSalary(5900);
28.
29.     List<Employee> empList = new ArrayList<Employee>();
30.     empList.add(employee1);
31.     empList.add(employee2);
32.     empList.add(employee3);
33.
34.     Department department = new Department();
35.     department.setName("Agriculture");
36.     department.setLocation("Gunipalli");
37.     department.setEmployeeList(empList);
38.
39.     session.save(department);
40.
41.     session.getTransaction().commit();
42.     SessionUtil.closeSession(session);
43. }
44. }
```

OUTPUT TABLES:

EMP				DEPT		
EID	NAME	SAL	E_DNO	DNO	NAME	LOC
1	kesavareddy	4600	1			
2	yellowreddy	5800	1			
3	cherry	5900	1	1	Agriculture	Gunipalli

RetriveDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import java.util.List;
4.
5. import org.hibernate.Session;
6.
7. import com.sekharit.hibernate.entity.Department;
8. import com.sekharit.hibernate.entity.Employee;
9. import com.sekharit.hibernate.util.SessionUtil;
10.
11. public class RetriveDAO {
12.     public static void main(String[] args) {
13.         Session session = SessionUtil.getSession();
14.         session.beginTransaction();
15.
16.         Department department = (Department) session.get(Department.class, 1);
17.         List<Employee> empList = department.getEmployeeList();
18.         sop("Department details are...");
19.         sop("Dno : " + department.getDno());
20.         sop("Name : " + department.getName());
21.         sop("Location : " + department.getLocation());
```

```
22.
23.         sop("\nEmployee details are... ");
24.         for (Employee employee : empList) {
25.             sop("\nEno : " + employee.getEno());
26.             sop("Name : " + employee.getName());
27.             sop("Salary : " + employee.getSalary());
28.         }
29.
30.         session.getTransaction().commit();
31.         SessionUtil.closeSession(session);
32.     }
33.
34.     public static void sop(Object object) {
35.         System.out.println(object);
36.     }
37. }
```

OUTPUT:

Department details are...

Dno : 1
Name : Agriculture
Location : Gunipalli

Eno : 1
Name : kesavareddy
Salary : 4600.0

Eno : 2
Name : yellareddy
Salary : 5800.0

Eno : 3
Name : cherry
Salary : 5900.0

UpadateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import java.util.Iterator;
4. import java.util.List;
5.
6. import org.hibernate.Session;
7.
8. import com.sekharit.hibernate.entity.Department;
9. import com.sekharit.hibernate.entity.Employee;
10. import com.sekharit.hibernate.util.SessionUtil;
11.
12. public class UpdateDAO {
13.     public static void main(String[] args) {
14.         Session session = SessionUtil.getSession();
15.         session.beginTransaction();
16.
17.         Department department = (Department) session.get(Department.class, 1);
18.         department.setName("IT");
19.         List<Employee> empList = department.getEmployeeList();
20.     }
21. }
```

```

21.         Iterator<Employee> iterator = empList.iterator();
22.         while (iterator.hasNext()) {
23.             Employee employee = iterator.next();
24.             if (employee.getSalary() > 5200) {
25.                 employee.setSalary(employee.getSalary() + 1000);
26.             }
27.         }
28.
29.         session.getTransaction().commit();
30.         SessionUtil.closeSession(session);
31.     }
32. }
```

OUTPUT TABLES:

EMP

ENO	NAME	SAL	E_DNO
1	kesavareddy	4600	1
2	yellareddy	6800	1
3	cherry	6900	1

DEPT

DNO	NAME	LOC
IT	Gunipalli	

DeleteDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.util.SessionUtil;
7.
8. public class DeleteDAO {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Department department = (Department) session.get(Department.class, 1);
14.         session.delete(department);
15.
16.         session.getTransaction().commit();
17.         SessionUtil.closeSession(session);
18.     }
19. }
```

OUTPUT TABLES:

EMP

ENO	NAME	SAL	E_DNO

DEPT

DNO	NAME	LOC

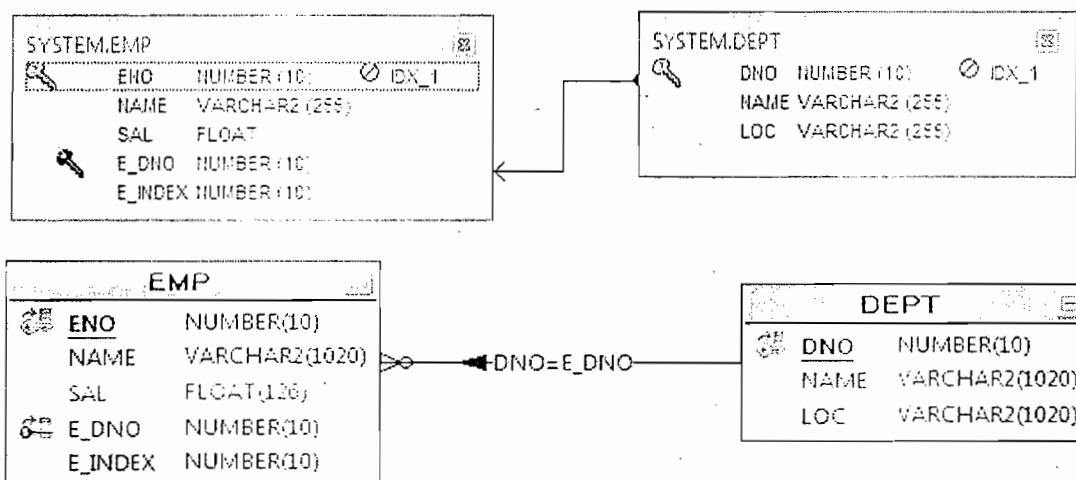
One-to-many(array) Unidirectional(Department to Employee)

```

    .
    .
    .
    > one2manyarrayuni
        .
        .
        .
        > src
            .
            .
            .
            > com.sekharit.hibernate.config
                .
                .
                .
            > com.sekharit.hibernate.dao
                .
                .
                .
                .
            > com.sekharit.hibernate.entity
                .
                .
                .
            > com.sekharit.hibernate.mapping
                .
                .
                .
            > com.sekharit.hibernate.util
                .
                .
            > JRE System Library
                .
                .
            > Referenced Libraries
                .
                .
                > ojdbc14.jar

```

Relation between the tables is like the following:

Department.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.CascadeType;
4. import javax.persistence.Column;
5. import javax.persistence.Entity;
6. import javax.persistence.FetchType;
7. import javax.persistence.GeneratedValue;

```

```
8. import javax.persistence.GenerationType;
9. import javax.persistence.Id;
10. import javax.persistence.JoinColumn;
11. import javax.persistence.OneToMany;
12. import javax.persistence.Table;
13.
14. import org.hibernate.annotations.GenericGenerator;
15. import org.hibernate.annotations.IndexColumn;
16.
17. @Entity
18. @Table(name = "DEPT")
19. public class Department {
20.     @Id
21.         @GenericGenerator(name = "myGenerator", strategy = "increment")
22.         @GeneratedValue(strategy=GenerationType.AUTO, generator="myGenerator")
23.         @Column(name = "DNO")
24.         private int dno;
25.         @Column(name = "NAME")
26.         private String name;
27.         @Column(name = "LOC")
28.         private String location;
29.         @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
30.         @JoinColumn(name = "E_DNO", referencedColumnName = "DNO")
31.         @IndexColumn(name = "E_INDEX")
32.         private Employee[] empArray;
33.
34.     //getters and setters
35. }
```

Employee.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.Table;
9.
10. import org.hibernate.annotations.GenericGenerator;
11.
12. @Entity
13. @Table(name = "EMP")
14. public class Employee {
15.     @Id
16.         @GenericGenerator(name = "myGenerator", strategy = "increment")
17.         @GeneratedValue(strategy=GenerationType.AUTO, generator="myGenerator")
18.         @Column(name = "ENO")
19.         private int eno;
20.         @Column(name = "NAME")
21.         private String name;
22.         @Column(name = "SAL")
23.         private double salary;
24.
25.
```

```
26.     //getters and setters  
27. }
```

Department.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>  
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >  
4. <hibernate-mapping>  
5.   <class name="com.sekharit.hibernate.entity.Department" table="DEPT">  
6.     <id name="dno" column="DNO">  
7.       <generator class="increment" />  
8.     </id>  
9.     <property name="name" column="NAME" />  
10.    <property name="location" column="LOC" />  
11.    <array name="empArray" cascade="all">  
12.      <key column="E_DNO" />  
13.      <index column="E_INDEX"></index>  
14.      <one-to-many class="com.sekharit.hibernate.entity.Employee" />  
15.    </array>  
16.  </class>  
17. </hibernate-mapping>
```

Employee.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>  
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >  
4. <hibernate-mapping>  
5.   <class name="com.sekharit.hibernate.entity.Employee" table="EMP">  
6.     <id column="ENO" name="eno">  
7.       <generator class="increment" />  
8.     </id>  
9.     <property column="NAME" name="name" />  
10.    <property column="SAL" name="salary" />  
11.  </class>  
12. </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;  
2.  
3. import org.hibernate.Session;  
4.  
5. import com.sekharit.hibernate.entity.Department;  
6. import com.sekharit.hibernate.entity.Employee;  
7. import com.sekharit.hibernate.util.SessionUtil;  
8.  
9. public class CreateDAO {  
10.    public static void main(String[] args) {  
11.        Session session = SessionUtil.getSession();  
12.        session.beginTransaction();  
13.  
14.        Employee employee1 = new Employee();  
15.        employee1.setName("somu");  
16.        employee1.setSalary(25000);
```

```

17.
18.     Employee employee2 = new Employee();
19.     employee2.setName("sekhar");
20.     employee2.setSalary(35000);
21.
22.     Employee employee3 = new Employee();
23.     employee3.setName("somasekhar");
24.     employee3.setSalary(60000);
25.
26.     Employee[] empArray = new Employee[]
27.                         { employee1, employee2, employee3 };
28.     Department department = new Department();
29.     department.setName("Accounts");
30.     department.setLocation("Anantapur");
31.     department.setEmpArray(empArray);
32.
33.     session.save(department);
34.
35.     session.getTransaction().commit();
36.     SessionUtil.closeSession(session);
37. }
38. }
```

OUTPUT TABLES:

EMP				DEPT			
EHO	NAME	SAL	E_DNO	E_INDEX	DNO	NAME	LOC
1 somu		25000	1	0			
2 sekhar		35000	1	1			
3 somasekhar		60000	1	2	1 Accounts	Anantapur	

RetriveDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.entity.Employee;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class RetriveDAO {
10.     public static void main(String[] args) {
11.         Session session = SessionUtil.getSession();
12.         session.beginTransaction();
13.
14.         Department department=(Department)session.get(Department.class,1);
15.         Employee[] employees = department.getEmpArray();
16.         sop("Department details are...\"");
17.         sop("Dno : " + department.getDno());
18.         sop("Name : " + department.getName());
19.         sop("Location : " + department.getLocation());
```

```
20.
21.         sop("\nEmployee details are... ");
22.         for (Employee employee : employees) {
23.             sop("\nEno : " + employee.getEno());
24.             sop("Name : " + employee.getName());
25.             sop("Salary : " + employee.getSalary());
26.         }
27.         session.getTransaction().commit();
28.         SessionUtil.closeSession(session);
29.     }
30.
31.     public static void sop(Object object) {
32.         System.out.println(object);
33.     }
34. }
```

OUTPUT:

Department details are...

Dno : 1
Name : Accounts
Location : Anantapur

Employee details are...

Eno : 1
Name : somu
Salary : 25000.0

Eno : 2
Name : sekhar
Salary : 35000.0

Eno : 3
Name : somasekhar
Salary : 60000.0

UpadateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.entity.Employee;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class UpdateDAO {
10.     public static void main(String[] args) {
11.         Session session = SessionUtil.getSession();
12.         session.beginTransaction();
13.
14.         Department department = (Department) session.get(Department.class, 1);
15.         department.setLocation("Hyderabad");
16.         Employee[] empArray = department.getEmpArray();
17.         for (Employee employee : empArray) {
18.             if (employee.getSalary() > 30000) {
19.                 employee.setName("sekharreddy");
```

```

20.          }
21.      }
22.      session.getTransaction().commit();
23.      SessionUtil.closeSession(session);
24.  }
25. }
```

OUTPUT TABLES:

EMP

ENO	NAME	SAL	E_DNO	E_INDEX
1	seenu	25000	1	0
2	sekharreddy	35000	1	1
3	sekharreddy	60000	1	2

DEPT

DNO	NAME	LOC
1	Accounts	Hyderabad

DeleteDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.util.SessionUtil;
7.
8. public class DeleteDAO {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Department department = (Department) session.get(Department.class, 1);
14.         session.delete(department);
15.
16.         session.getTransaction().commit();
17.         SessionUtil.closeSession(session);
18.     }
19. }
```

OUTPUT TABLES:

EMP

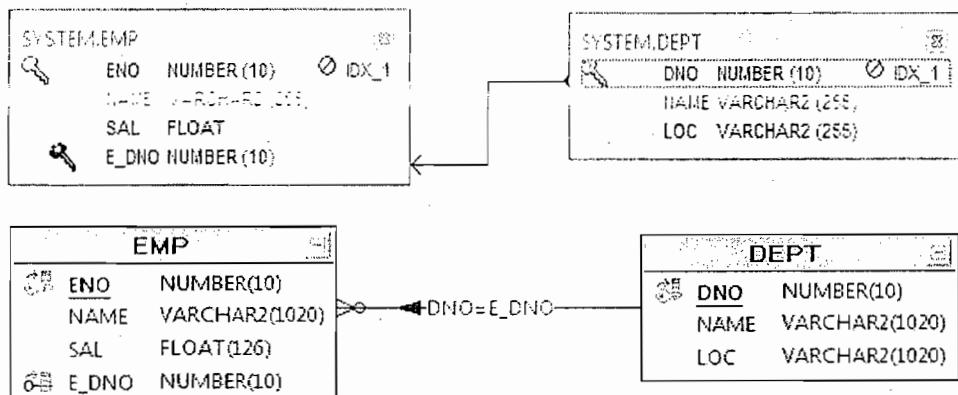
ENO	NAME	SAL	E_DNO	E_INDEX
1	seenu	25000	1	0

DEPT

DNO	NAME	LOC
1	Accounts	Hyderabad

Many-to-one(unidirectional)(Employee to Department)

Relation between the tables is like the following:

Department.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.Table;
9.

```

```
10. import org.hibernate.annotations.GenericGenerator;
11.
12. import javax.persistence.*;
13. @Table(name = "DEPT")
14. public class Department {
15.     @Id
16.     @GenericGenerator(name = "myGenerator", strategy = "increment")
17.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
18.     @Column(name = "DNO")
19.     private int dno;
20.     @Column(name = "NAME")
21.     private String name;
22.     @Column(name = "LOC")
23.     private String location;
24.
25.     //getters and setters
26. }
```

Employee.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.CascadeType;
4. import javax.persistence.Column;
5. import javax.persistence.Entity;
6. import javax.persistence.FetchType;
7. import javax.persistence.GeneratedValue;
8. import javax.persistence.GenerationType;
9. import javax.persistence.Id;
10. import javax.persistence.JoinColumn;
11. import javax.persistence.ManyToOne;
12. import javax.persistence.Table;
13.
14. import org.hibernate.annotations.GenericGenerator;
15.
16. @Entity
17. @Table(name = "EMP")
18. public class Employee {
19.     @Id
20.     @GenericGenerator(name = "myGenerator", strategy = "increment")
21.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
22.     @Column(name = "ENO")
23.     private int eno;
24.     @Column(name = "NAME")
25.     private String name;
26.     @Column(name = "SAL")
27.     private double salary;
28.     @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
29.     @JoinColumn(name = "E_DNO", referencedColumnName = "DNO")
30.     private Department department;
31.
32.     //getters and setters
33. }
```

Department.hbm.xml

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

```
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Department" table="DEPT">
6.     <id name="dno" column="DNO">
7.       <generator class="increment" />
8.     </id>
9.     <property name="name" column="NAME" />
10.    <property name="location" column="LOC" />
11.  </class>
12. </hibernate-mapping>
```

Employee.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Employee" table="EMP">
6.     <id column="ENO" name="eno">
7.       <generator class="increment" />
8.     </id>
9.     <property column="NAME" name="name" />
10.    <property column="SAL" name="salary" />
11.    <many-to-one name="department"
12.      class="com.sekharit.hibernate.entity.Department"
13.      cascade="all"
14.      lazy="false"
15.      column="E_DNO"></many-to-one>
16.  </class>
17. </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.entity.Employee;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class CreateDAO {
10.     public static void main(String[] args) {
11.         Session session = SessionUtil.getSession();
12.         session.beginTransaction();
13.
14.         Department department = new Department();
15.         department.setName("Agriculture");
16.         department.setLocation("Gunipalli");
17.
18.         Employee employee1 = new Employee();
19.         employee1.setName("kesavareddy");
20.         employee1.setSalary(4600);
21.         employee1.setDepartment(department);
```

```

22.
23.     Employee employee2 = new Employee();
24.     employee2.setName("yellareddy");
25.     employee2.setSalary(5800);
26.     employee2.setDepartment(department);
27.
28.     Employee employee3 = new Employee();
29.     employee3.setName("cherry");
30.     employee3.setSalary(5900);
31.     employee3.setDepartment(department);
32.
33.     session.save(employee1);
34.     session.save(employee2);
35.     session.save(employee3);
36.
37.     session.getTransaction().commit();
38.     SessionUtil.closeSession(session);
39. }
40. }
```

OUTPUT TABLES:

EMP			
ENO	NAME	SAL	E_DNO
1	kesavareddy	4600	1
2	yellareddy	5800	1
3	cherry	5900	1

DEPT			
DNO	NAME	LOC	
1	Agriculture	Gunipalli	

RetriveDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.entity.Employee;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class RetriveDAO {
10.     public static void main(String[] args) {
11.         Session session = SessionUtil.getSession();
12.         session.beginTransaction();
13.
14.         Employee employee = (Employee) session.get(Employee.class, 1);
15.         Department department = employee.getDepartment();
16.         sop("Department details.....");
17.         sop("Dno : " + department.getDno());
18.         sop("Name : " + department.getName());
19.         sop("Location : " + department.getLocation());
20.         sop("\nEmployee details of '" + department.getName()
21.             + "' department");
22.         sop("Eno : " + employee.getEno());
23.         sop("Name : " + employee.getName());
```

```
24.         sop("Salary : " + employee.getSalary());
25.
26.         session.getTransaction().commit();
27.         SessionUtil.closeSession(session);
28.     }
29.
30.     public static void sop(Object object) {
31.         System.out.println(object);
32.     }
33. }
34.
```

OUTPUT:

Department details are...

Dno : 1
Name : Agriculture
Location : Gunipalli

Employee details of 'Agriculture' department

Eno : 1
Name : kesavareddy
Salary : 4600.0

UpdateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.entity.Employee;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class UpdateDAO {
10.     public static void main(String[] args) {
11.         Session session = SessionUtil.getSession();
12.         session.beginTransaction();
13.
14.         Employee employee = (Employee) session.get(Employee.class, 2);
15.         Department department = employee.getDepartment();
16.         employee.setName("yerragudi");
17.         department.setLocation("hyd");
18.
19.         session.getTransaction().commit();
20.         SessionUtil.closeSession(session);
21.     }
22. }
```

OUTPUT TABLES:

EMP

E_NO	NAME	SAL	E_DNO
1	kesavareddy	4600	1
2	yerragudi	5800	1
3	cherry	5900	1

DEPT

DNO	LOC	NAME
1	hyd	Agriculture

DeleteDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3.
4. import com.sekharit.hibernate.entity.Employee;
5. import com.sekharit.hibernate.util.SessionUtil;
6. public class DeleteDAO {
7.     public static void main(String[] args) {
8.         Session session=SessionUtil.getSession();
9.         session.beginTransaction();
10.
11.         Employee employee1=(Employee)session.get(Employee.class, 3);
12.         session.delete(employee1);
13.
14.         Employee employee2=(Employee)session.get(Employee.class, 2);
15.         session.delete(employee2);
16.
17.         Employee employee3=(Employee)session.get(Employee.class, 1);
18.         session.delete(employee3);
19.
20.         session.getTransaction().commit();
21.         SessionUtil.closeSession(session);
22.     }
23. }
```

OUTPUT TABLES:

EMP				DEPT		
E_NO	NAME	SAL	E_DNO	DNO	NAME	LOC
1	ABC	10000	2	1	HR	Bangalore

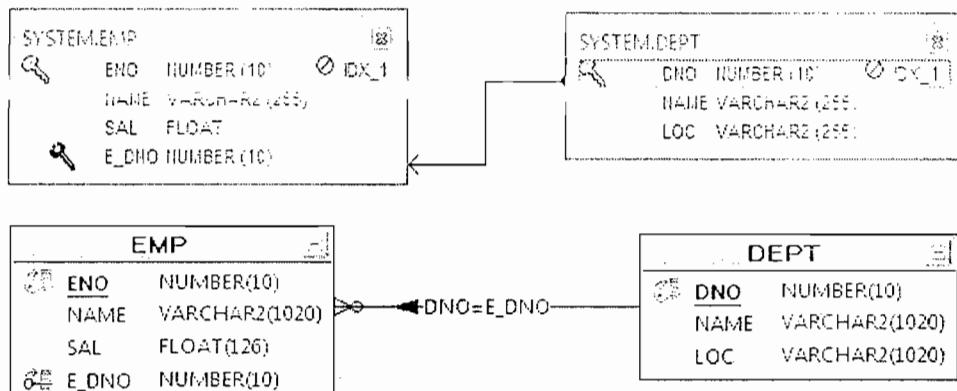
One-to-Many & Many-to-One Bidirectional(from Employee to Department)

```

4. com.onemany2many2onebi
  5. src
    6. com.sekharit.hibernate.config
      7. hibernate.cfg.xml
    8. com.sekharit.hibernate.dao
      9. CreateDAO1.java
      10. CreateDAO2.java
      11. DeleteDAO1.java
      12. DeleteDAO2.java
      13. RetrieveDAO1.java
      14. RetrieveDAO2.java
      15. UpdateDAO1.java
      16. UpdateDAO2.java
  17. com.sekharit.hibernate.entity
    18. Department.java
    19. Employee.java
  20. com.sekharit.hibernate.mapping
    21. Department.hbm.xml
    22. Employee.hbm.xml
  23. com.sekharit.hibernate.util
    24. SessionUtil.java
  25. JRE System Library [jre6]
  26. Hibernate 4.x Libraries
  27. Referenced Libraries

```

Relation between tables is like the following:



Department.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import java.util.List;
4.
5. import javax.persistence.CascadeType;
6. import javax.persistence.Column;
7. import javax.persistence.Entity;
8. import javax.persistence.FetchType;
9. import javax.persistence.GeneratedValue;
10. import javax.persistence.GenerationType;
11. import javax.persistence.Id;
12. import javax.persistence.JoinColumn;
13. import javax.persistence.OneToMany;
14. import javax.persistence.Table;

```

```
15.  
16. import org.hibernate.annotations.GenericGenerator;  
17.  
18. @Entity  
19. @Table(name = "DEPT")  
20. public class Department {  
21.     @Id  
22.         @GenericGenerator(name = "myGenerator", strategy = "increment")  
23.         @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")  
24.         @Column(name = "DNO")  
25.         private int dno;  
26.         @Column(name = "NAME")  
27.         private String name;  
28.         @Column(name = "LOC")  
29.         private String location;  
30.         @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)  
31.             @JoinColumn(name = "E_DNO", referencedColumnName = "DNO")  
32.             private List<Employee> employeeList;  
33.  
34.     //getters and setters  
35. }
```

Employee.java

```
1. package com.sekharit.hibernate.entity;  
2.  
3. import javax.persistence.CascadeType;  
4. import javax.persistence.Column;  
5. import javax.persistence.Entity;  
6. import javax.persistence.FetchType;  
7. import javax.persistence.GeneratedValue;  
8. import javax.persistence.GenerationType;  
9. import javax.persistence.Id;  
10. import javax.persistence.JoinColumn;  
11. import javax.persistence.ManyToOne;  
12. import javax.persistence.Table;  
13.  
14. import org.hibernate.annotations.GenericGenerator;  
15.  
16. @Entity  
17. @Table(name = "EMP")  
18. public class Employee {  
19.     @Id  
20.         @GenericGenerator(name = "myGenerator", strategy = "increment")  
21.         @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")  
22.         @Column(name = "ENO")  
23.         private int eno;  
24.         @Column(name = "NAME")  
25.         private String name;  
26.         @Column(name = "SAL")  
27.         private double salary;  
28.         @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)  
29.             @JoinColumn(name = "E_DNO", referencedColumnName = "DNO")  
30.             private Department department;  
31.  
32.     //getters and setters
```

33. }

Department.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Department" table="DEPT">
6.     <id name="dno" column="DNO">
7.       <generator class="increment" />
8.     </id>
9.     <property name="name" column="NAME" />
10.    <property name="location" column="LOC" />
11.    <bag name="employeeList" cascade="all" lazy="false">
12.      <key column="E_DNO" />
13.      <one-to-many class="com.sekharit.hibernate.entity.Employee" />
14.    </bag>
15.  </class>
16. </hibernate-mapping>
```

Employee.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Employee" table="EMP">
6.     <id column="ENO" name="eno">
7.       <generator class="increment" />
8.     </id>
9.     <property column="NAME" name="name" />
10.    <property column="SAL" name="salary" />
11.    <many-to-one name="department"
12.      class="com.sekharit.hibernate.entity.Department"
13.      cascade="all"
14.      lazy="false"
15.      column="E_DNO"></many-to-one>
16.  </class>
17. </hibernate-mapping>
```

CreateDAO1.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.entity.Employee;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class CreateDAO1 {
10.     public static void main(String[] args) {
11.         Session session = SessionUtil.getSession();
12.         session.beginTransaction();
13.     }
14. }
```

```

14.         Department department = new Department();
15.         department.setName("Agriculture");
16.         department.setLocation("Gunipalli");
17.
18.         Employee employee1 = new Employee();
19.         employee1.setName("kesavareddy");
20.         employee1.setSalary(4600);
21.         employee1.setDepartment(department);
22.
23.         Employee employee2 = new Employee();
24.         employee2.setName("yellareddy");
25.         employee2.setSalary(5800);
26.         employee2.setDepartment(department);
27.
28.         Employee employee3 = new Employee();
29.         employee3.setName("cherry");
30.         employee3.setSalary(5900);
31.         employee3.setDepartment(department);
32.
33.         session.save(employee1);
34.         session.save(employee2);
35.         session.save(employee3);
36.
37.         session.getTransaction().commit();
38.         SessionUtil.closeSession(session);
39.
40.     }

```

OUTPUT TABLES:

EMP

DEPT

ENO	NAME	SAL	E_DNO
1	kesavareddy	4600	1
2	yellareddy	5800	1
3	cherry	5900	1

DNO	NAME	LOC
1	Agriculture	Gunipalli

RetriveDAO1.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.entity.Employee;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class RetriveDAO1 {
10.     public static void main(String[] args) {
11.         Session session = SessionUtil.getSession();
12.         session.beginTransaction();
13.
14.         Employee employee = (Employee) session.get(Employee.class, 1);
15.         Department department = employee.getDepartment();

```

```

16.         sop("Department details.....");
17.         sop("Dno : " + department.getDno());
18.         sop("Name : " + department.getName());
19.         sop("Location : " + department.getLocation());
20.         sop("\nEmployee details of '" + department.getName()
21.                           + "' department");
22.         sop("Eno : " + employee.getEno());
23.         sop("Name : " + employee.getName());
24.         sop("Salary : " + employee.getSalary());
25.
26.         session.getTransaction().commit();
27.         SessionUtil.closeSession(session);
28.     }
29.
30.     public static void sop(Object object) {
31.         System.out.println(object);
32.     }
33. }
```

OUTPUT:

Department details are...

Dno : 1
 Name : Agriculture
 Location : Gunipalli

Employee details of 'Agriculture' department

Eno : 1
 Name : kesavareddy
 Salary : 4600.0

UpadateDAO1.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3.
4. import com.sekharit.hibernate.entity.Department;
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.util.SessionUtil;
7. public class UpadateDAO1 {
8.     public static void main(String[] args) {
9.         Session session=SessionUtil.getSession();
10.        session.beginTransaction();
11.
12.        Employee employee = (Employee) session.get(Employee.class, 2);
13.        Department department = employee.getDepartment();
14.        employee.setName("yerragudi");
15.        department.setLocation("hyd");
16.
17.        session.getTransaction().commit();
18.        SessionUtil.closeSession(session);
19.    }
20. }
```

OUTPUT TABLES:

EMP

DEPT

ENO	NAME	SAL	E_DNO
1	kesavareddy	4600	1
2	yerragudi	5800	1
3	cherry	5900	1

DNO	LOC	NAME
hyd	Agriculture	

DeleteDAO1.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.util.SessionUtil;
7.
8. public class DeleteDAO1 {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Employee employee1 = (Employee) session.get(Employee.class, 3);
14.         Employee employee2 = (Employee) session.get(Employee.class, 2);
15.         Employee employee3 = (Employee) session.get(Employee.class, 1);
16.
17.         session.delete(employee1);
18.         session.delete(employee2);
19.         session.delete(employee3);
20.
21.         session.getTransaction().commit();
22.         SessionUtil.closeSession(session);
23.     }
24. }
25.
```

OUTPUT TABLES:

EMP				DEPT		
ENO	NAME	SAL	E_DNO	DNO	NAME	LOC
1	kesavareddy	4600	1	hyd	Agriculture	

CreateDAO2.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. import org.hibernate.Session;
7.
8. import com.sekharit.hibernate.entity.Department;
9. import com.sekharit.hibernate.entity.Employee;
10. import com.sekharit.hibernate.util.SessionUtil;
11.
12. public class CreateDAO2 {
13.     public static void main(String[] args) {

```

```

14.         Session session = SessionUtil.getSession();
15.         session.beginTransaction();
16.
17.         Employee employee1 = new Employee();
18.         employee1.setName("kesavareddy");
19.         employee1.setSalary(4600);
20.
21.         Employee employee2 = new Employee();
22.         employee2.setName("yellareddy");
23.         employee2.setSalary(5800);
24.
25.         Employee employee3 = new Employee();
26.         employee3.setName("cherry");
27.         employee3.setSalary(5900);
28.
29.         List<Employee> empList = new ArrayList<Employee>();
30.         empList.add(employee1);
31.         empList.add(employee2);
32.         empList.add(employee3);
33.
34.         Department department = new Department();
35.         department.setName("Agriculture");
36.         department.setLocation("Gunipalli");
37.         department.setEmployeeList(empList);
38.
39.         session.save(department);
40.
41.         session.getTransaction().commit();
42.         SessionUtil.closeSession(session);
43.     }
44. }
```

OUTPUT TABLES:

EMP

DEPT

ENO	NAME	SAL	E_DNO
1	kesavareddy	4600	1
2	yellareddy	5800	1
3	cherry	5900	1

DNO	NAME	LOC
1	Agriculture	Gunipalli

RetriveDAO2.java

```

1. package com.sekharit.hibernate.dao;
2. import java.util.List;
3. import org.hibernate.Session;
4. import com.sekharit.hibernate.entity.Department;
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.util.SessionUtil;
7.
8. public class RetriveDAO2 {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
```

```
12.
13.     Department department = (Department) session.get(Department.class, 1);
14.     List<Employee> empList = department.getEmployeeList();
15.     sop("Department details are... ");
16.     sop("Dno : " + department.getDno());
17.     sop("Name : " + department.getName());
18.     sop("Location : " + department.getLocation());
19.
20.     sop("\nEmployee details are... ");
21.     for (Employee employee : empList) {
22.         sop("Eno : " + employee.getEno());
23.         sop("Name : " + employee.getName());
24.         sop("Salary : " + employee.getSalary());
25.     }
26.
27.     session.getTransaction().commit();
28.     SessionUtil.closeSession(session);
29. }
30.
31. public static void sop(Object object) {
32.     System.out.println(object);
33. }
34. }
```

OUTPUT:

Department details are...

Dno : 1
Name : Agriculture
Location : Gunipalli

Eno : 1
Name : kesavareddy
Salary : 4600.0

Eno : 2
Name : yellareddy
Salary : 5800.0

Eno : 3
Name : cherry
Salary : 5900.0

UpdateDAO2.java

```
1. package com.sekharit.hibernate.dao;
2. import java.util.Iterator;
3. import java.util.List;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.entity.Department;
6. import com.sekharit.hibernate.entity.Employee;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class UpdateDAO2 {
10.     public static void main(String[] args) {
11.         Session session = SessionUtil.getSession();
12.         session.beginTransaction();
13.     }
14. }
```

```

14.     Department department = (Department) session.get(Department.class, 1);
15.     department.setName("IT");
16.     List<Employee> empList = department.getEmployeeList();
17.
18.     Iterator<Employee> iterator = empList.iterator();
19.     while (iterator.hasNext()) {
20.         Employee employee = iterator.next();
21.         if (employee.getSalary() > 5200) {
22.             employee.setSalary(employee.getSalary() + 1000);
23.         }
24.     }
25.
26.     session.getTransaction().commit();
27.     SessionUtil.closeSession(session);
28. }
29.

```

OUTPUT TABLES:

EMP

DEPT

E_NO	NAME	SAL	E_DNO
1	kesavareddy	4600	1
2	yellareddy	6800	1
3	cherry	6900	1

DNO	NAME	LOC
1	IT	Gunipalli

DeleteDAO.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Department;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class DeleteDAO2 {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.        Department department = (Department) session.get(Department.class, 1);
11.        session.delete(department);
12.
13.        session.getTransaction().commit();
14.        SessionUtil.closeSession(session);
15.    }
16. }

```

OUTPUT TABLES:

EMP

DEPT

E_NO	NAME	SAL	E_DNO
1	kesavareddy	4600	1

DNO	NAME	LOC
1	IT	Gunipalli

Many-to-Many Unidirectional

- ⇒ If we consider the relationship between STUDENT and COURSE with the rule, student can join in multiple courses and a course can contain multiple students. We can implement this with many-to-many relationship.
- ⇒ many-to-many is not possible with single table or two tables. We need minimum three tables to implement many-to-many relationship.
- ⇒ Tables are

STUDENT	
SNO[p.k]	SNAME
101	sachin
102	dravid
103	srikanth
104	ganguly

STUDENT_COURSE	
SNO[f.k]	CNO[f.k]
101	2001
101	2003
101	2004
104	2002
102	2002
103	2002

COURSE	
CNO[p.k]	CNAME
2001	JAVA
2002	.NET
2003	PHP
2004	GATE

The screens may be like this

Student Registration Screen

SNO :	<input type="text"/>
Name :	<input type="text"/>
Age :	<input type="text"/>
Qualification:	<input type="text"/>
Course :	JAVA : <input type="checkbox"/> .NET : <input type="checkbox"/> PHP : <input type="checkbox"/> GATE : <input type="checkbox"/>
<input type="button" value="Submit"/>	

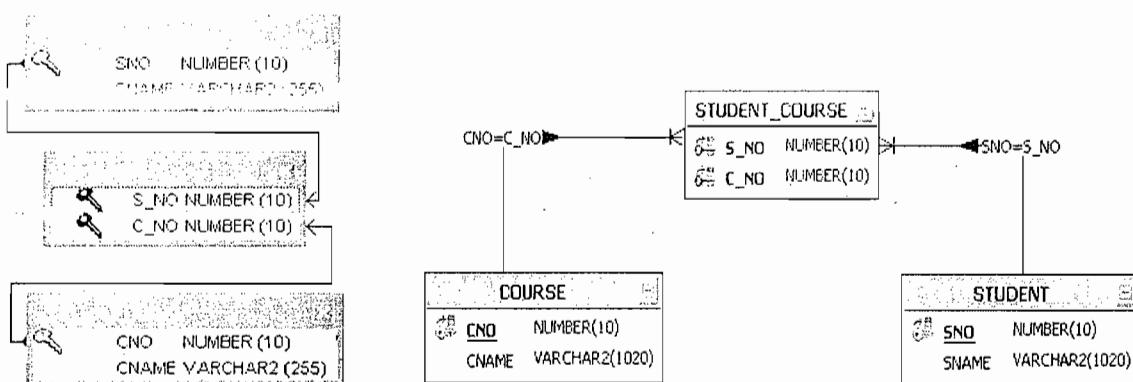
- ⇒ Like this each student has the option to join in multiple courses, and each course has the possibility to have multiple students.
- ⇒ We never implement hibernate code first. First database design has to be done. And first client has to give the screens of the project, Based on the database design and screens, java people will decide what relationships need to configure among entities.

```

- (.) ManyToManyUnibag
  - (.) src
    - (.) com.sekharit.hibernate.config
      hibernate.cfg.xml
    - (.) com.sekharit.hibernate.dao
      + (.) CreateDAO.java
      + (.) DeleteDAO.java
      + (.) RetrieveDAO.java
      + (.) UpdateDAO.java
    - (.) com.sekharit.hibernate.mapping
      + (.) Course.hbm.xml
      + (.) Student.hbm.xml
    - (.) com.sekharit.hibernate.model
      + (.) Course.java
      + (.) Student.java
    - (.) com.sekharit.hibernate.util
      + (.) SessionUtil.java
+ (.) JRE System Library [jre6]
- (.) Referenced Libraries
  + (.) ojdbc14.jar - 10.2
+ (.) Hibernate 4.x Library

```

Relation between the tables is like the following:



Course.java

```

1. package com.sekharit.hibernate.model;
2. import javax.persistence.Column;
3. import javax.persistence.Entity;
4. import javax.persistence.GeneratedValue;
5. import javax.persistence.GenerationType;
6. import javax.persistence.Id;
7. import javax.persistence.Table;
8. import org.hibernate.annotations.GenericGenerator;
9. @Entity
10. @Table(name = "COURSE")
11. public class Course {
12.     @Id
13.     @GenericGenerator(name = "myGenerator", strategy = "increment")

```

```
14.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
15.     @Column(name = "CNO")
16.     private int cno;
17.     @Column(name = "CNAME")
18.     private String cname;
19.
20.     //getters & setters
21.
22. }
```

Student.java

```
1. package com.sekharit.hibernate.model;
2. import java.util.List;
3. import javax.persistence.CascadeType;
4. import javax.persistence.Column;
5. import javax.persistence.Entity;
6. import javax.persistence.FetchType;
7. import javax.persistence.GeneratedValue;
8. import javax.persistence.GenerationType;
9. import javax.persistence.Id;
10. import javax.persistence.JoinColumn;
11. import javax.persistence.JoinTable;
12. import javax.persistence.ManyToMany;
13. import javax.persistence.Table;
14. import org.hibernate.annotations.GenericGenerator;
15. import org.hibernate.annotations.IndexColumn;
16. @Entity
17. @Table(name = "STUDENT")
18. public class Student {
19.     @Id
20.     @GenericGenerator(name = "myGenerator", strategy = "increment")
21.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
22.     @Column(name = "SNO")
23.     private int sno;
24.     @Column(name = "SNAME")
25.     private String sname;
26.     @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
27.     @JoinTable(name = "STUDENT_COURSE", joinColumns = {
28.         @JoinColumn(name = "S_NO", referencedColumnName = "SNO") },
29.         inverseJoinColumns = {
30.             @JoinColumn(name = "C_NO", referencedColumnName = "CNO") })
31.     private List<Course> clist;
32.
33.     //getters & setters
34. }
```

Student.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.model.Student" table="STUDENT">
6.     <id name="sno" column="NO">
7.       <generator class="increment"></generator>
8.     </id>
```

```
9.      <property name="sname" column="NAME"></property>
10.     <bag name="clist" table="STUDENT_COURSE" cascade="all" lazy="true"
11.         fetch="join">
12.         <key column="S_SNO"></key>
13.         <many-to-many class "com.sekharit.hibernate.model.Course"
14.             column="C_CNO" />
15.     </bag>
16.   </class>
17. </hibernate-mapping>
```

Course.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4.   <hibernate-mapping>
5.     <class name="com.sekharit.hibernate.model.Course" table="COURSE">
6.       <id name="cno" column="NO">
7.         <generator class="increment"></generator>
8.       </id>
9.       <property name="cname" column="NAME"></property>
10.    </class>
11.  </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import java.util.ArrayList;
3. import java.util.List;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class CreateDAO {
9.   public static void main(String[] args) {
10.     Session session = SessionUtil.getSession();
11.     session.beginTransaction();
12.
13.     Course c1 = new Course();
14.     Course c2 = new Course();
15.
16.     c1.setCname("SPRING");
17.     c2.setCname("HTIBERNATE");
18.
19.     List<Course> cList = new ArrayList<Course>();
20.     cList.add(c1);
21.     cList.add(c2);
22.
23.     Student s1 = new Student();
24.     s1.setSname("SEKHAR");
25.     s1.setClist(cList);
26.     session.save(s1);
27.
28.     Student s2 = new Student();
29.     s2.setSname("SOMU");
30.     s2.setClist(cList);
31.     session.save(s2);
```

```
32.  
33.         session.getTransaction().commit();  
34.         session.close();  
35.     }  
36. }
```

OUTPUT TABLES

STUDENT		COURSE		STUDENT_COURSE	
SNO	SNAME	CNO	CNAME	S_NO	C_NO
1	SEHARIT	1	SPRING	1	1
2	SOMU	2	HIBERNATE	2	2

RetriveDAO.java

```
1. package com.sekharit.hibernate.dao;  
2. import java.util.Iterator;  
3. import java.util.List;  
4. import org.hibernate.Session;  
5. import com.sekharit.hibernate.model.Course;  
6. import com.sekharit.hibernate.model.Student;  
7. import com.sekharit.hibernate.util.SessionUtil;  
8. public class RetriveDAO {  
9.     public static void main(String[] args) {  
10.         Session session = SessionUtil.getSession();  
11.         session.beginTransaction();  
12.           
13.         Student student = (Student) session.get(Student.class, 1);  
14.         sop(student.getSno());  
15.         sop(student.getSname());  
16.           
17.         List<Course> cList = student.getClist();  
18.         Iterator<Course> iterator = cList.iterator();  
19.         while (iterator.hasNext()) {  
20.             Course course = (Course) iterator.next();  
21.             sop("Course No: "+course.getCno());  
22.             sop("Course Name: "+course.getCname());  
23.         }  
24.           
25.         session.getTransaction().commit();  
26.         session.close();  
27.     }  
28.       
29.     public static void sop(Object object) {  
30.         System.out.println(object);  
31.     }  
32. }
```

OUTPUT

Course No: 1
Course Name: SPRING
Course No: 2
Course Name: HIBERNATE

UpdateDAO.java

```
1. package com.sekharit.hibernate.dao;
```

```

2. import java.util.Iterator;
3. import java.util.List;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class UpdateDAO {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction().begin();
12.         Student student = (Student) session.get(Student.class, 2);
13.         student.setSname("SOMASEKHAR");
14.
15.         List<Course> cSet = student.getClist();
16.         Iterator<Course> iterator = cSet.iterator();
17.         while (iterator.hasNext()) {
18.             Course course = iterator.next();
19.             if (course.getCname().equals("HIBERNATE")) {
20.                 course.setCname("EJB");
21.             }
22.         }
23.
24.         session.getTransaction().commit();
25.         session.close();
26.     }
27. }
```

OUTPUT TABLES

STUDENT		COURSE		STUDENT_COURSE	
SNO	SNAME	CNO	CNAME	S_NO	C_NO
1	SEKHAR	1	SPRING	1	1
2	SOMASEKHAR	2	EJB	2	2

DeleteDAO.java

```

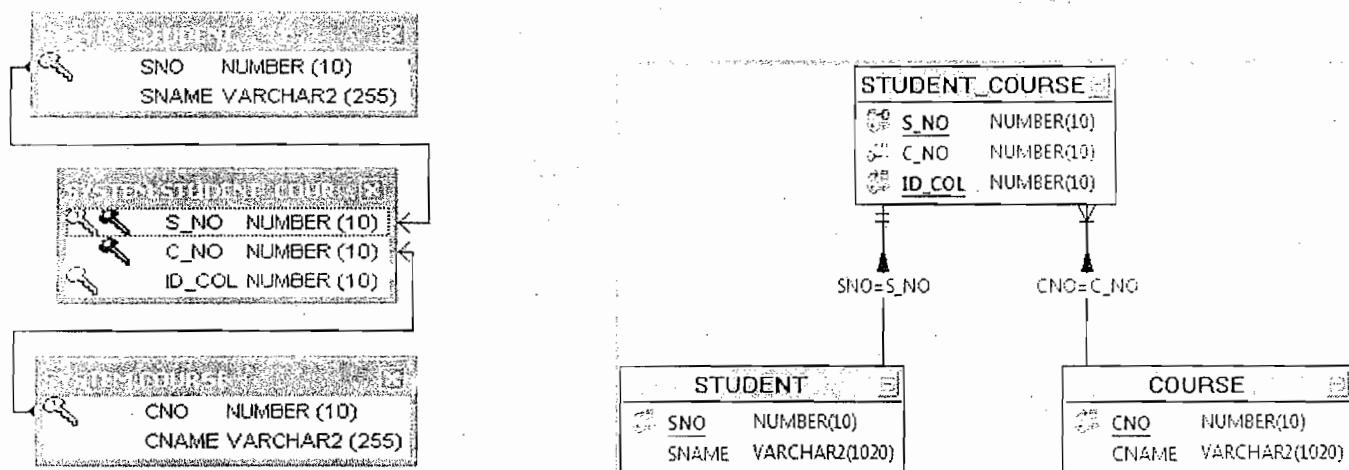
1. package com.sekharit.hibernate.dao;
2. import java.util.List;
3. import org.hibernate.Session;
4. import com.sekharit.hibernate.model.Course;
5. import com.sekharit.hibernate.model.Student;
6. import com.sekharit.hibernate.util.SessionUtil;
7. public class DeleteDAO {
8.     public static void main(String[] args) {
9.         Session session = SessionUtil.getSession();
10.        session.beginTransaction();
11.
12.        Student student = (Student) session.get(Student.class, 1);
13.
14.        List<Course> cList = student.getClist();
15.        cList.remove(0);
16.
17.        session.getTransaction().commit();
18.        session.close();
19.    }
20. }
```

OUTPUT TABLES

STUDENT		COURSE		STUDENT_COURSE	
SNO	SNAME	CNO	CNAME	S_NO	C_NO
1	SEKHAR	1	SPRING	2	1
2	SOMASEKHAR	2	EJB	2	2
				1	2

- ManyToManyUnitList
- src
 - com.sekharit.hibernate.config
 - ↳ hibernate.cfg.xml
 - com.sekharit.hibernate.dao
 - + CreateDAO.java
 - + DeleteDAO.java
 - + RetrieveDAO.java
 - + UpdateDAO.java
 - com.sekharit.hibernate.mapping
 - ↳ Course.hbm.xml
 - ↳ Student.hbm.xml
 - com.sekharit.hibernate.model
 - + Course.java
 - + Student.java
 - com.sekharit.hibernate.util
 - + SessionUtil.java
- + JRE System Library [jres]
- Referenced Libraries
 - + ojdbc14.jar - 51
- + Hibernate 4.x Library

Relation between the tables is like the following:



Course.java

```

1. package com.sekharit.hibernate.model;
2. import javax.persistence.Column;
3. import javax.persistence.Entity;
4. import javax.persistence.GeneratedValue;

```

```
5. import javax.persistence.GenerationType;
6. import javax.persistence.Id;
7. import javax.persistence.Table;
8. import org.hibernate.annotations.GenericGenerator;
9.     @Entity
10.    @Table(name = "COURSE")
11.   public class Course {
12.       @Id
13.       @GenericGenerator(name = "myGenerator", strategy = "increment")
14.       @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
15.       @Column(name = "CNO")
16.       private int cno;
17.       @Column(name = "CNAME")
18.       private String cname;
19.
20.       //getters & setters
21.
22.   }
```

Student.java

```
1. package com.sekharit.hibernate.model;
2. import java.util.List;
3. import javax.persistence.CascadeType;
4. import javax.persistence.Column;
5. import javax.persistence.Entity;
6. import javax.persistence.FetchType;
7. import javax.persistence.GeneratedValue;
8. import javax.persistence.GenerationType;
9. import javax.persistence.Id;
10. import javax.persistence.JoinColumn;
11. import javax.persistence.JoinTable;
12. import javax.persistence.ManyToMany;
13. import javax.persistence.Table;
14. import org.hibernate.annotations.GenericGenerator;
15. import org.hibernate.annotations.IndexColumn;
16.     @Entity
17.     @Table(name = "STUDENT")
18.   public class Student {
19.       @Id
20.       @GenericGenerator(name = "myGenerator", strategy = "increment")
21.       @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
22.       @Column(name = "SNO")
23.       private int sno;
24.       @Column(name = "SNAME")
25.       private String sname;
26.       @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
27.       @JoinTable(name = "STUDENT_COURSE", joinColumns = {
28.           @JoinColumn(name = "S_NO", referencedColumnName = "SNO") },
29.           inverseJoinColumns = {
30.               @JoinColumn(name = "C_NO", referencedColumnName = "CNO") })
31.       private List<Course> clist;
32.
33.       //getters & setters
34.   }
```

Student.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4.   <hibernate-mapping>
5.     <class name="com.sekharit.hibernate.model.Student" table="STUDENT">
6.       <id name="sno" column="NO">
7.         <generator class="increment"></generator>
8.       </id>
9.       <property name="sname" column="NAME"></property>
10.      <list name="clist" table="STUDENT_COURSE" cascade="all"
11.        lazy="true" fetch="join" >
12.          <key column="S_SNO"></key>
13.          <index column="ID_COL"></index>
14.          <many-to-many class="com.sekharit.hibernate.model.Course"
15.            column="C_CNO" />
16.        </list>
17.      </hibernate-mapping>
```

Course.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4.   <hibernate-mapping>
5.     <class name="com.sekharit.hibernate.model.Course" table="COURSE">
6.       <id name="cno" column="NO">
7.         <generator class="increment"></generator>
8.       </id>
9.       <property name="cname" column="NAME"></property>
10.      </class>
11.    </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import java.util.ArrayList;
3. import java.util.List;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class CreateDAO {
9.   public static void main(String[] args) {
10.     Session session = SessionUtil.getSession();
11.     session.beginTransaction();
12.
13.     Course c1 = new Course();
14.     Course c2 = new Course();
15.     c1.setCname("SPRING");
16.     c2.setCname("HIBERNATE");
17.
18.     List<Course> cList = new ArrayList<Course>();
19.     cList.add(c1);
20.     cList.add(c2);
21. }
```

```

22.         Student s1 = new Student();
23.         s1.setSname("SEKHAR");
24.         s1.setClist(cList);
25.
26.         session.save(s1);
27.
28.         Student s2 = new Student();
29.         s2.setSname("SOMU");
30.         s2.setClist(cList);
31.         session.save(s2);
32.
33.         session.getTransaction().commit();
34.         session.close();
35.     }
36. }
```

OUTPUT TABLES

STUDENT		COURSE		STUDENT_COURSE		
SNO	SNAME	CNO	CNAME	S_NO	C_NO	ID_COL
1	SEKHAR	1	SPRING	1	1	0
2	SOMU	2	HIBERNATE	1	2	1
				2	1	0
				2	2	1

RetriveDAO.java

```

1. package com.sekharit.hibernate.dao;
2. import java.util.Iterator;
3. import java.util.List;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class RetrieverDAO {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Student student = (Student) session.get(Student.class, 1);
14.         sop(student.getSno());
15.         sop(student.getSname());
16.
17.         List<Course> cList = student.getClist();
18.         Iterator<Course> iterator = cList.iterator();
19.         while (iterator.hasNext()) {
20.             Course course = (Course) iterator.next();
21.             sop("Course No: "+course.getCno());
22.             sop("Course Name: "+course.getCname());
23.         }
24.
25.         session.getTransaction().commit();
26.         session.close();
27.     }
28.
29.     public static void sop(Object object) {
30.         System.out.println(object);
31.     }
}
```

32. }

OUTPUT

Course No: 1
Course Name: SPRING
Course No: 2
Course Name: HIBERNATE

UpdateDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import java.util.List;
3. import org.hibernate.Session;
4. import com.sekharit.hibernate.model.Course;
5. import com.sekharit.hibernate.model.Student;
6. import com.sekharit.hibernate.util.SessionUtil;
7. public class UpdateDAO {
8.     public static void main(String[] args) {
9.         Session session = SessionUtil.getSession();
10.        session.beginTransaction();
11.
12.        Student student = (Student) session.get(Student.class, 2);
13.        List<Course> courses = student.getClist();
14.        Course course = new Course();
15.        course.setCname("ADV.JAVA");
16.        courses.add(course);
17.
18.        session.getTransaction().commit();
19.        session.close();
20.    }
21. }
```

OUTPUT TABLES

STUDENT		COURSE	STUDENT_COURSE		
SNO	SNAME	NO NAME	S_SNO	C_CNO	ID_COL
1	SEKHAR	1 SPRING	1	1	0
2	SOMU	2 HIBERNATE	2	2	1
		3 ADV.JAVA	2	3	2

DeleteDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import java.util.List;
3. import org.hibernate.Session;
4. import com.sekharit.hibernate.model.Course;
5. import com.sekharit.hibernate.model.Student;
6. import com.sekharit.hibernate.util.SessionUtil;
7. public class DeleteDAO {
8.     public static void main(String[] args) {
9.         Session session = SessionUtil.getSession();
10.        session.beginTransaction();
11.
12.        Student student = (Student) session.get(Student.class, 1);
13.
14.        List<Course> cList = student.getClist();
```

```

15.         cList.remove(0);
16.
17.         session.getTransaction().commit();
18.         session.close();
19.     }
20. }
```

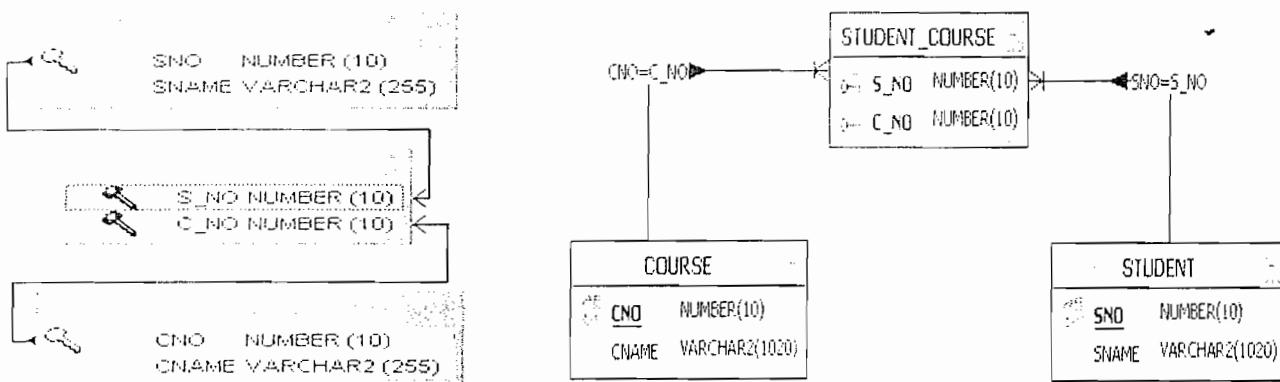
OUTPUT TABLES

STUDENT		COURSE	STUDENT_COURSE		
SNO	SNAME	NO NAME	S_SNO	C_CNO	ID_COL
1	SPRING	1 SPRING	2	1	0
2	HIBERNATE	2 HIBERNATE	2	2	1
2	SOMU	3 ADV.JAVA	2	3	2

↳ ManyToManyUnset

- ↳ src
 - ↳ com.sekharit.hibernate.config
 - ↳ hibernate.cfg.xml
 - ↳ com.sekharit.hibernate.dao
 - ↳ CreateDAO.java
 - ↳ DeleteDAO.java
 - ↳ RetrieveDAO.java
 - ↳ UpdateDAO.java
 - ↳ com.sekharit.hibernate.mapping
 - ↳ Course.hbm.xml
 - ↳ Student.hbm.xml
 - ↳ com.sekharit.hibernate.model
 - ↳ Course.java
 - ↳ Student.java
 - ↳ com.sekharit.hibernate.util
 - ↳ SessionUtil.java
- ↳ JRE System Library [pre]
 - ↳ Hibernate 4.x Libraries
- ↳ Referenced Libraries
 - ↳ ojdbc14.jar

Relation between the tables is like the following:



Course.java

```

1. package com.sekharit.hibernate.model;
2. import javax.persistence.Column;
3. import javax.persistence.Entity;
4. import javax.persistence.GeneratedValue;
5. import javax.persistence.GenerationType;
6. import javax.persistence.Id;
7. import javax.persistence.Table;
8. import org.hibernate.annotations.GenericGenerator;
9.     @Entity
10.    @Table(name = "COURSE")
11.    public class Course {
12.        @Id
13.            @GenericGenerator(name = "myGenerator", strategy = "increment")
14.            @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
15.            @Column(name = "CNO")
16.            private int cno;
17.            @Column(name = "CNAME")
18.            private String cname;
19.
20.        // getters & setters
21.    }

```

Student.java

```

1. package com.sekharit.hibernate.model;
2. import java.util.Set;
3. import javax.persistence.CascadeType;
4. import javax.persistence.Column;
5. import javax.persistence.Entity;
6. import javax.persistence.FetchType;
7. import javax.persistence.GeneratedValue;
8. import javax.persistence.GenerationType;
9. import javax.persistence.Id;
10. import javax.persistence.JoinColumn;
11. import javax.persistence.JoinTable;
12. import javax.persistence.ManyToMany;
13. import javax.persistence.Table;
14. import org.hibernate.annotations.GenericGenerator;
15.     @Entity
16.    @Table(name = "STUDENT")
17.    public class Student {
18.        @Id

```

```
19.     @GenericGenerator(name = "myGenerator", strategy = "increment")
20.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
21.     @Column(name = "SNO")
22.     private int sno;
23.     @Column(name = "SNAME")
24.     private String sname;
25.     @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
26.     @JoinTable(name = "STUDENT_COURSE", joinColumns = {
27.         @JoinColumn(name = "S_NO", referencedColumnName = "SNO")
28.     },
29.         inverseJoinColumns = {
30.             @JoinColumn(name = "C_NO", referencedColumnName = "CNO")
31.         })
32.     private Set<Course> cset;
33.
34.     //getters & setters
35. }
```

Student.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4.   <hibernate-mapping>
5.     <class name="com.sekharit.hibernate.model.Student" table="STUDENT">
6.       <id name="sno" column="NC">
7.         <generator class="increment"></generator>
8.       </id>
9.       <property name="sname" column="NAME"></property>
10.      <set name="cset" table="STUDENT_COURSE" cascade="all"
11.        lazy="true" fetch="join" >
12.        <key column="S_SNO"></key>
13.        <many-to-many class="com.sekharit.hibernate.model.Course"
14.                      column="C_CNO" >
15.          </many-to-many>
16.        </set>
17.      </class>
18.   </hibernate-mapping>
```

Course.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4.   <hibernate-mapping>
5.     <class name="com.sekharit.hibernate.model.Course" table="COURSE">
6.       <id name="cno" column="NO">
7.         <generator class="increment"></generator>
8.       </id>
9.       <property name="cname" column="NAME"></property>
10.      </class>
11.   </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import java.util.HashSet;
3. import java.util.Set;
```

```

4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class CreateDAO {
9.     public static void main(String[] args) {
10.         Session session=SessionUtil.getSession();
11.         session.beginTransaction().begin();
12.         Course c1=new Course();
13.         c1.setCname("HIBERNATE");
14.
15.         Course c2=new Course();
16.         c2.setCname("SPRING");
17.
18.         Set<Course> cSet=new HashSet<Course>();
19.         cSet.add(c1);
20.         cSet.add(c2);
21.
22.         Student s1=new Student();
23.         s1.setSname("SOMU");
24.
25.         Student s2=new Student();
26.         s2.setSname("SEKHAR");
27.
28.         s1.setCset(cSet);
29.         s2.setCset(cSet);
30.
31.         session.save(s1);
32.         session.save(s2);
33.
34.         session.getTransaction().commit();
35.         session.close();
36.     }
37. }
38. }
```

OUTPUT TABLES

STUDENT	COURSE	STUDENT_COURSE	
SNO	SNAME	S_SNO	C_CNO
1	SEKHAR	1	1
2	SOMU	1	2
		2	1
		2	2

RetriveDAO.java

```

1. package com.sekharit.hibernate.dao;
2. import java.util.Iterator;
3. import java.util.Set;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class RetriveDAO {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
```

```

11.
12.         Student student = (Student) session.get(Student.class, 1);
13.         sop("Student No: "+student.getSno());
14.         sop("Student Name: "+student.getSname());
15.
16.         Set<Course> cSet = student.getCset();
17.         Iterator<Course> iterator = cSet.iterator();
18.         while (iterator.hasNext()) {
19.             Course course = (Course) iterator.next();
20.             sop("Course No: " + course.getCno());
21.             sop("Course Name: " + course.getCname());
22.         }
23.
24.         session.close();
25.     }
26.
27.     public static void sop(Object object) {
28.         System.out.println(object);
29.     }
30. }
```

OUTPUT

Student No: 1
 Student Name: SOMU
 Course No: 1
 Course Name: SPRING
 Course No: 2
 Course Name: HIBERNATE

UpdateDAO.java

```

1. package com.sekharit.hibernate.dao;
2. import java.util.Iterator;
3. import java.util.Set;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class UpdateDAO {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Student student = (Student) session.get(Student.class, 2);
14.         student.setSname("SOMASEKHAR");
15.
16.         Set<Course> cSet = student.getCset();
17.         Iterator<Course> iterator = cSet.iterator();
18.         while (iterator.hasNext()) {
19.             Course course = iterator.next();
20.             if (course.getCname().equals("HIBERNATE")) {
21.                 course.setCname("EJB");
22.             }
23.         }
24.
25.         session.getTransaction().commit();
```

```

26.         session.close();
27.     }
28. }
```

OUTPUT TABLES

STUDENT		COURSE		STUDENT_COURSE	
SNO	SNAME	CNO	CNAME	S_SNO	C_CNO
1	SOMU	1	SPRING	1	1
2	SOMASEKHAR	2	EJB	2	2

DeleteDAO.java

```

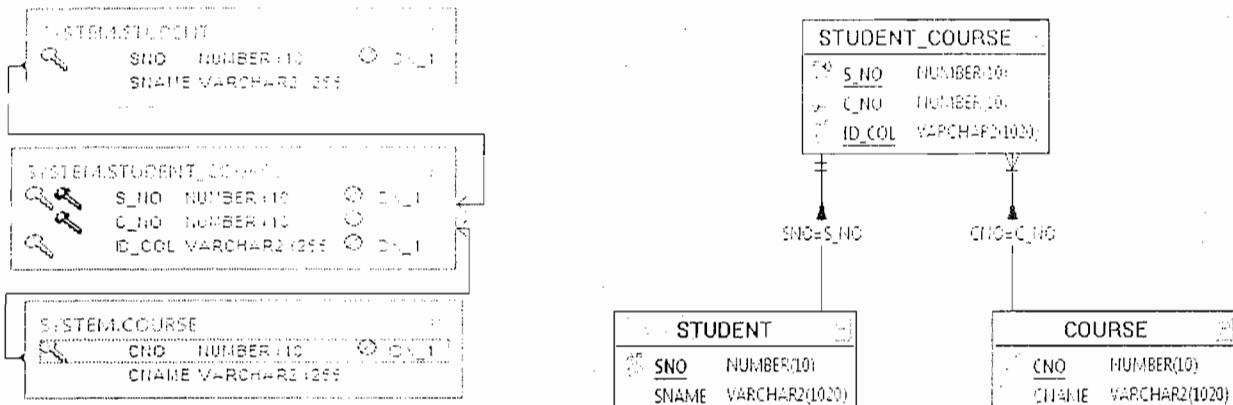
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.model.Student;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class DeleteDAO {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction().begin();
9.
10.        Student student = (Student) session.get(Student.class, 1);
11.        student.setCset(null);
12.        session.delete(student);
13.
14.        session.getTransaction().commit();
15.        session.close();
16.    }
17. }
```

OUTPUT TABLES

STUDENT		COURSE		STUDENT_COURSE	
SNO	SNAME	CNO	CNAME	S_SNO	C_CNO
1	SOMU	1	SPRING	2	1
2	SOMASEKHAR	2	EJB	2	2

ManyToManyunimap

Relation between the tables is like the following:



Course.java

```

1. package com.sekharit.hibernate.model;
2. import javax.persistence.Column;
3. import javax.persistence.Entity;
4. import javax.persistence.GeneratedValue;
5. import javax.persistence.GenerationType;
6. import javax.persistence.Id;
7. import javax.persistence.Table;
8. import org.hibernate.annotations.GenericGenerator;
9. @Entity
10. @Table(name = "COURSE")
11. public class Course {
12.     @Id
13.     @GenericGenerator(name = "myGenerator", strategy = "increment")
14.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
15.     @Column(name = "CNO")
16.     private int cno;
17.     @Column(name = "CNAME")
18.     private String cname;
19.
20.     //getters & setters
21. }
  
```

Student.java

```

1. package com.sekharit.hibernate.model;
2. import java.util.Set;
3. import javax.persistence.CascadeType;
4. import javax.persistence.Column;
5. import javax.persistence.Entity;
6. import javax.persistence.FetchType;
7. import javax.persistence.GeneratedValue;
8. import javax.persistence.GenerationType;
9. import javax.persistence.Id;
10. import javax.persistence.JoinColumn;
11. import javax.persistence.JoinTable;
12. import javax.persistence.ManyToMany;
13. import javax.persistence.Table;
14. import org.hibernate.annotations.GenericGenerator;
  
```

```
15. @Entity
16. @Table(name = "STUDENT")
17. public class Student {
18.     @Id
19.         @GenericGenerator(name = "myGenerator", strategy = "increment")
20.         @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
21.         @Column(name = "SNO")
22.         private int sno;
23.         @Column(name = "SNAME")
24.         private String sname;
25.         @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
26.         @JoinTable(name = "STUDENT_COURSE",
27.             joinColumns = {@JoinColumn(name = "S_NO", referencedColumnName = "SNO") },
28.             inverseJoinColumns = {
29.                 @JoinColumn(name = "C_NO", referencedColumnName = "CNO") })
30.         @MapKey(name = "ID_COL")
31.         private Map<String, Course> cmap;
32.
33.         //getters & setters
34. }
```

Student.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.     <class name="com.sekharit.hibernate.model.Student" table="STUDENT">
6.         <id name="sno" column="NO">
7.             <generator class="increment"></generator>
8.         </id>
9.         <property name="sname" column="NAME"></property>
10.        <map name="cmap" table="STUDENT_COURSE" cascade="all">
11.            <index column="ID_COL" type="string"></index>
12.            <many-to-many class="com.sekharit.hibernate.entity.Course"
13.                column="C_NO">
14.            </many-to-many>
15.        </map>
16.    </class>
17. </hibernate-mapping>
```

Course.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.     <class name="com.sekharit.hibernate.model.Course" table="COURSE">
6.         <id name="cno" column="NO">
7.             <generator class="increment"></generator>
8.         </id>
9.         <property name="cname" column="NAME"></property>
10.    </class>
11. </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;
```

```

2. import java.util.HashMap;
3. import java.util.Map;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.entity.Course;
6. import com.sekharit.hibernate.entity.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class CreateDAO {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Course c1 = new Course();
14.         c1.setCname("SPRING");
15.
16.         Course c2 = new Course();
17.         c2.setCname("HIBERNATE");
18.
19.         Map<String, Course> map = new HashMap<String, Course>();
20.         map.put("spring", c1);
21.         map.put("hibernate", c2);
22.
23.         Student s1 = new Student();
24.         s1.setSname("sekhar");
25.         s1.setCmap(map);
26.
27.         Student s2 = new Student();
28.         s2.setSname("somu");
29.         s2.setCmap(map);
30.
31.         session.save(s2);
32.         session.save(s1);
33.
34.         session.getTransaction().commit();
35.         session.close();
36.     }
37. }

```

OUTPUT TABLES

STUDENT		COURSE	STUDENT_COURSE			
SNO	NAME	NO	NAME	S_SNO	C_NO	ID_COL
1	somu	1	HIBERNATE	1	1	hibernate
2	sekhar	2	SPRING	1	2	spring
				2	1	hibernate
				2	2	spring

RetriveDAO.java

```

1. package com.sekharit.hibernate.dao;
2. import java.util.Iterator;
3. import java.util.Map;
4. import java.util.Set;
5. import org.hibernate.Session;
6. import com.sekharit.hibernate.entity.Course;
7. import com.sekharit.hibernate.entity.Student;

```

```
8. import com.sekharit.hibernate.util.SessionUtil;
9. public class RetrieveDAO {
10.     public static void main(String[] args) {
11.         Session session = SessionUtil.getSession();
12.         session.beginTransaction().begin();
13.
14.         Student student = (Student) session.get(Student.class, 2);
15.         sop("Student No: "+student.getSno());
16.         sop("Student Name: "+student.getSname());
17.
18.         Map<String, Course> map = student.getCmap();
19.         Set<String> set = map.keySet();
20.         Iterator<String> iterator = set.iterator();
21.         while (iterator.hasNext()) {
22.             String key = iterator.next();
23.             Course course = (Course) map.get(key);
24.             sop("Course No: " +course.getCno());
25.             sop("Course Name: " +course.getCname());
26.         }
27.
28.         session.getTransaction().commit();
29.         session.close();
30.     }
31.
32.     public static void sop(Object object) {
33.         System.out.println(object);
34.     }
35. }
```

OUTPUT

```
Student No: 1
Student Name: somu
Course No: 1
Course Name: HIBERNATE
Course No: 2
Course Name: SPRING
```

UpadateDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import java.util.Iterator;
3. import java.util.Map;
4. import java.util.Set;
5. import org.hibernate.Session;
6. import com.sekharit.hibernate.entity.Course;
7. import com.sekharit.hibernate.entity.Student;
8. import com.sekharit.hibernate.util.SessionUtil;
9. public class UpdateDAO {
10.     public static void main(String[] args) {
11.         Session session = SessionUtil.getSession();
12.         session.beginTransaction().begin();
13.
14.         Student student = (Student) session.get(Student.class, 2);
15.
16.         Map<String, Course> map = student.getCmap();
17.         Set<String> set = map.keySet();
```

```
18.         Iterator<String> iterator = set.iterator();
19.         while (iterator.hasNext()) {
20.             String key = iterator.next();
21.             if (key.equals("spring")) {
22.                 Course course = (Course) map.get(key);
23.                 course.setCname("ANNOSPRING");
24.             }
25.         }
26.
27.         session.getTransaction().commit();
28.         session.close();
29.
30.     }
```

OUTPUT TABLES

STUDENT	COURSE	STUDENT_COURSE
S_SNO	C_NO	ID_COL
NO NAME	NO NAME	1 1 hibernate
1 somu	1 HIBERNATE	1 2 spring
2 sekhar	2 ANNOSPRING	2 1 hibernate
		2 2 spring

DeleteDAO.java

```
1. package com.sekharit.hibernate.dao;
2. import java.util.Map;
3. import org.hibernate.Session;
4. import com.sekharit.hibernate.entity.Course;
5. import com.sekharit.hibernate.entity.Student;
6. import com.sekharit.hibernate.util.SessionUtil;
7. public class DeleteDAO {
8.     public static void main(String[] args) {
9.         Session session = SessionUtil.getSession();
10.        session.beginTransaction().begin();
11.
12.        Student student = (Student) session.get(Student.class, 1);
13.        Map<String, Course> map = student.getCmap();
14.        map.remove("hibernate");
15.
16.        session.getTransaction().commit();
17.        session.close();
18.    }
19.
20. }
```

OUTPUT TABLES

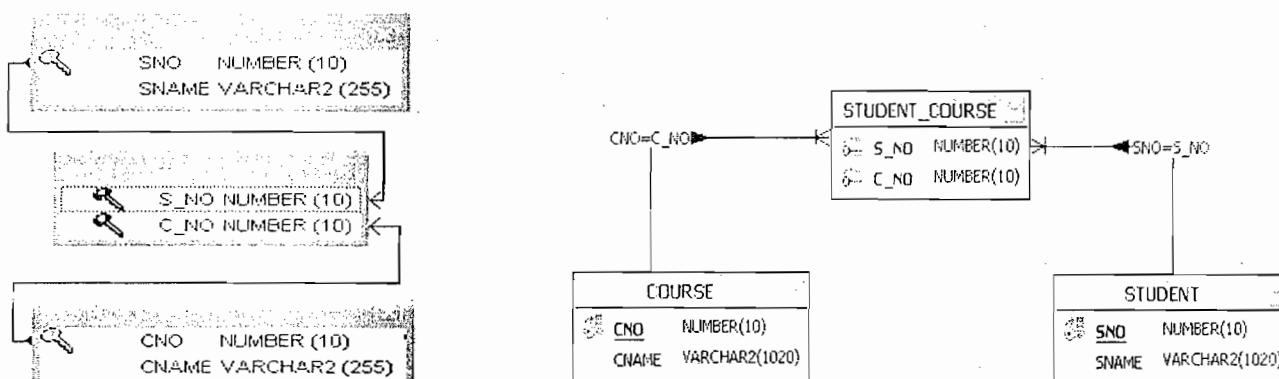
STUDENT	COURSE	STUDENT_COURSE
S_SNO	C_NO	ID_COL
NO NAME	NO NAME	1 2 spring
1 somu	1 HIBERNATE	2 1 hibernate
2 sekhar	2 ANNOSPRING	2 2 spring

Hibernate-'has-a' relationship mismatch

By Mr. SekharReddy

```
↳ ManyToManyBiset
  ↳ src
    ↳ com.sekharit.hibernate.config
      ↳ hibernate.cfg.xml
    ↳ com.sekharit.hibernate.dao
      ↳ CreateDAO1.java
      ↳ CreateDAO2.java
      ↳ DeleteDAO1.java
      ↳ DeleteDAO2.java
      ↳ RetrieveDAO1.java
      ↳ RetrieveDAO2.java
      ↳ UpdateDAO1.java
      ↳ UpdateDAO2.java
    ↳ com.sekharit.hibernate.mapping
      ↳ Course.hbm.xml
      ↳ Student.hbm.xml
    ↳ com.sekharit.hibernate.model
      ↳ Course.java
      ↳ Student.java
    ↳ com.sekharit.hibernate.util
      ↳ SessionUtil.java
  ↳ JRE System Library
  ↳ Hibernate 4.0 Libraries
  ↳ Referenced Libraries
    ↳ ojdbc14.jar
```

Relation between the tables is like the following:



Course.java

```
1. package com.sekharit.hibernate.model;
2. import javax.persistence.Column;
3. import javax.persistence.Entity;
4. import javax.persistence.GeneratedValue;
5. import javax.persistence.GenerationType;
6. import javax.persistence.Id;
7. import javax.persistence.Table;
8. import org.hibernate.annotations.GenericGenerator;
9.   @Entity
10.  @Table(name = "COURSE")
11.  public class Course {
12.    @Id
13.    @GenericGenerator(name = "myGenerator", strategy = "increment")
14.    @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
15.    @Column(name = "CNO")
16.    private int cno;
17.    @Column(name = "CNAME")
18.    private String cname;
19.    @ManyToMany(cascade=CCascadeType.ALL, fetch=FetchType.LAZY)
```

```
20.     @JoinTable(name = "STUDENT_COURSE",
21.         joinColumns = {
22.             @JoinColumn(name = "C_NO", referencedColumnName = "CNO") },
23.         inverseJoinColumns = {
24.             @JoinColumn(name = "S_NO", referencedColumnName = "SNO") })
25.     private Set<Student> studentSet;
26.
27.     //getters & setters
28. }
```

Student.java

```
1. package com.sekharit.hibernate.model;
2. import java.util.Set;
3. import javax.persistence.CascadeType;
4. import javax.persistence.Column;
5. import javax.persistence.Entity;
6. import javax.persistence.FetchType;
7. import javax.persistence.GeneratedValue;
8. import javax.persistence.GenerationType;
9. import javax.persistence.Id;
10. import javax.persistence.JoinColumn;
11. import javax.persistence.JoinTable;
12. import javax.persistence.ManyToOne;
13. import javax.persistence.Table;
14. import org.hibernate.annotations.GenericGenerator;
15. @Entity
16. @Table(name = "STUDENT")
17. public class Student {
18.     @Id
19.         @GenericGenerator(name = "myGenerator", strategy = "increment")
20.         @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
21.         @Column(name = "SNO")
22.     private int sno;
23.     @Column(name = "SNAME")
24.     private String sname;
25.     @ManyToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
26.     @JoinTable(name = "STUDENT_COURSE", joinColumns = {
27.             @JoinColumn(name = "S_NO", referencedColumnName = "SNO")
28.         },
29.             inverseJoinColumns = {
30.                 @JoinColumn(name = "C_NO", referencedColumnName = "CNO")
31.             })
32.     private Set<Course> cset;
33.
34.     //getters & setters
35. }
```

Student.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.     <class name="com.sekharit.hibernate.model.Student" table="STUDENT">
6.         <id name="sno" column="NO">
7.             <generator class="increment"></generator>
```

```

8.      </id>
9.      <property name="sname" column="NAME"></property>
10.     <set name="cset" table="STUDENT_COURSE" cascade="all"
11.       lazy="true" fetch="join" >
12.         <key column="S_SNO"></key>
13.         <many-to-many class="com.sekharit.hibernate.model.Course"
14.           column="C_CNO" >
15.         </many-to-many>
16.       </set>
17.     </class>
18.   </hibernate-mapping>

```

Course.hbm.xml

```

1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4.   <hibernate-mapping>
5.     <class name="com.sekharit.hibernate.model.Course" table="COURSE">
6.       <id name="cno" column="NO">
7.         <generator class="increment"></generator>
8.       </id>
9.       <property name="cname" column="NAME"></property>
10.      <set name="studentSet" table="STUDENT_COURSE" cascade="all"
11.        lazy="true" fetch="join" >
12.          <key column="C_CNO"></key>
13.          <many-to-many class="com.sekharit.hibernate.model.Student"
14.            column="S_SNO" >
15.          </many-to-many>
16.        </set>
17.      </class>
18.    </hibernate-mapping>

```

CreateDAO1.java

```

1. package com.sekharit.hibernate.dao;
2. import java.util.HashSet;
3. import java.util.Set;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class CreateDAO1 {
9.   public static void main(String[] args) {
10.     Session session = SessionUtil.getSession();
11.     session.beginTransaction();
12.
13.     Course c1 = new Course();
14.     c1.setCname("HIBERNATE");
15.     Course c2 = new Course();
16.     c2.setCname("SPRING");
17.
18.     Set<Course> cSet = new HashSet<Course>();
19.     cSet.add(c1);
20.     cSet.add(c2);
21.

```

```
22.         Student s1 = new Student();
23.         s1.setSname("SEKHAR");
24.
25.         s1.setCset(cSet);
26.
27.         session.save(s1);
28.
29.         session.getTransaction().commit();
30.         session.close();
31.     }
32. }
```

OUTPUT TABLES

STUDENT		COURSE		STUDENT_COURSE	
SNO	SNAME	CNO	CNAME	S_SNO	C_CNO
1	SEKHAR	1	SPRING	1	1
		2	HIBERNATE	1	2

RetriveDAO1.java

```
1. package com.sekharit.hibernate.dao;
2. import java.util.Iterator;
3. import java.util.Set;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class RetrieverDAO1 {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Student student = (Student) session.get(Student.class, 1);
14.         sop("Student No: "+student.getSno());
15.         sop("Student Name: "+student.getSname());
16.
17.         Set<Course> cSet = student.getCset();
18.         Iterator<Course> iterator = cSet.iterator();
19.         while (iterator.hasNext()) {
20.             Course course = (Course) iterator.next();
21.             sop("Course No: " + course.getCno());
22.             sop("Course Name: " + course.getCname());
23.         }
24.
25.         session.getTransaction().commit();
26.         session.close();
27.     }
28.
29.     public static void sop(Object object) {
30.         System.out.println(object);
31.     }
32. }
```

OUTPUT

Student No: 1

```

Student Name: SEKHAR
Course No: 1
Course Name: SPRING
Course No: 2
Course Name: HIBERNATE

```

UpadateDAO1.java

```

1. package com.sekharit.hibernate.dao;
2. import java.util.Iterator;
3. import java.util.Set;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class UpadateDAO1 {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Student student = (Student) session.get(Student.class, 1);
14.         student.setSname("SOMASEKHAR");
15.         Set<Course> courses = student.getCset();
16.         Iterator<Course> iterator = courses.iterator();
17.         while (iterator.hasNext()) {
18.             Course course = (Course) iterator.next();
19.             if (course.getCname().equals("SPRING")) {
20.                 course.setCname("ANNOSPRING");
21.             }
22.         }
23.
24.         session.getTransaction().commit();
25.         session.close();
26.     }
27. }

```

OUTPUT TABLES

STUDENT		COURSE		STUDENT_COURSE	
SNO	SNAME	CNO	CNAME	S_NO	C_NO
1	SOMASEKHAR	1	ANNOSPRING	1	1
		2	HIBERNATE	1	2

DeleteDAO1.java

```

1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.model.Student;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class DeleteDAO1 {
6.     public static void main(String[] args) {
7.         Session session = SessionUtil.getSession();
8.         session.beginTransaction();
9.
10.         Student student = (Student) session.get(Student.class, 1);
11.         student.setCset(null);
12.

```

```

13.         session.getTransaction().commit();
14.         session.close();
15.     }
16. }
```

OUTPUT TABLES

STUDENT	COURSE	STUDENT_COURSE
SNO SNAME 1 SOMASEKHAR	CNO CNAME 1 ANNOSSPRING 2 HIBERNATE	S_SNO C_CNO

CreateDAO2.java

```

1. package com.sekharit.hibernate.dao;
2. import java.util.HashSet;
3. import java.util.Set;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class CreatedAO2 {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Student s1 = new Student();
14.         s1.setSname("SOMU");
15.
16.         Student s2 = new Student();
17.         s2.setSname("YERRAGUDI");
18.
19.         Set<Student> studentsSet = new HashSet<Student>();
20.         studentsSet.add(s1);
21.         studentsSet.add(s2);
22.
23.         Course c1 = new Course();
24.         c1.setCname("STRUTS");
25.         c1.setStudentSet(studentsSet);
26.
27.         session.save(c1);
28.
29.         session.getTransaction().commit();
30.         session.close();
31.     }
32. }
```

OUTPUT TABLES

STUDENT	COURSE	STUDENT_COURSE
SNO SNAME 1 SOMASEKHAR 2 SOMU 3 YERRAGUDI	CNO CNAME 1 ANNOSSPRING 2 HIBERNATE 3 STRUTS	S_NO C_NO

RetriveDAO2.java

```
1. package com.sekharit.hibernate.dao;
2. import java.util.Iterator;
3. import java.util.Set;
4. import org.hibernate.Session;
5. import com.sekharit.hibernate.model.Course;
6. import com.sekharit.hibernate.model.Student;
7. import com.sekharit.hibernate.util.SessionUtil;
8. public class RetrieverDAO2 {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         Course course = (Course) session.get(Course.class, 3);
14.         sop("Course No: "+course.getCno());
15.         sop("Course Name: "+course.getCname());
16.
17.         Set<Student> set = course.getStudentSet();
18.         Iterator<Student> iterator = set.iterator();
19.         while (iterator.hasNext()) {
20.             Student student = (Student) iterator.next();
21.             sop("Student No: " + student.getSno());
22.             sop("Student Name: " + student.getSname());
23.         }
24.
25.         session.getTransaction().commit();
26.         session.close();
27.     }
28.
29.     public static void sop(Object object) {
30.         System.out.println(object);
31.     }
32. }
```

OUTPUT

```
Course No: 3
Course Name: STRUTS
Student No: 2
Student Name: SOMU
Student No: 3
Student Name: YERRAGUDI
```

UpadateDAO2.java

```
1. package com.sekharit.hibernate.dao;
2. package com.sekharit.hibernate.dao;
3. import java.util.Iterator;
4. import java.util.Set;
5. import org.hibernate.Session;
6. import com.sekharit.hibernate.model.Course;
7. import com.sekharit.hibernate.model.Student;
8. import com.sekharit.hibernate.util.SessionUtil;
9. public class UpdateDAO2 {
10.     public static void main(String[] args) {
11.         Session session = SessionUtil.getSession();
12.         session.beginTransaction();
```

```

13.
14.     Course course = (Course) session.get(Course.class, 3);
15.     course.setCname("ANNOSTRUTS");
16.
17.     Set<Student> students = course.getStudentSet();
18.     Iterator<Student> iterator = students.iterator();
19.     while (iterator.hasNext()) {
20.         Student student = (Student) iterator.next();
21.         if (student.getSname().equals("SOMU")) {
22.             student.setSname("Y.SOMU");
23.         }
24.
25.
26.         session.getTransaction().commit();
27.         session.close();
28.     }
29. }
```

OUTPUT TABLES

STUDENT		COURSE		STUDENT_COURSE	
SNO	SNAME	CNO	CNAME	S_NO	C_NO
1	SOMASEKHAR	1	ANNOSPRING		
2	Y.SOMU	2	HIBERNATE	2	3
3	YERRAGUDI	3	ANNOSTRUTS	3	3

DeleteDAO2.java

```

1. package com.sekharit.hibernate.dao;
2. package com.sekharit.hibernate.dao;
3. import org.hibernate.Session;
4. import com.sekharit.hibernate.model.Course;
5. import com.sekharit.hibernate.util.SessionUtil;
6. public class DeleteDAO2 {
7.     public static void main(String[] args) {
8.         Session session = SessionUtil.getSession();
9.         session.beginTransaction();
10.
11.        Course course = (Course) session.get(Course.class, 3);
12.        course.setStudentSet(null);
13.
14.        session.getTransaction().commit();
15.        session.close();
16.    }
17. }
```

OUTPUT TABLES

STUDENT		COURSE		STUDENT_COURSE	
SNO	SNAME	CNO	CNAME	S_SNO	C_CNO
1	SOMASEKHAR	1	ANNOSPRING		
2	Y.SOMU	2	HIBERNATE		
3	YERRAGUDI	3	ANNOSTRUTS		

Inheritance mismatches

Q) Store SalariedEmployee, HourlyEmployee details into database ?

```
class SalariedEmployee{  
    int eno;  
    String ename;  
    Double salary;  
    // getters & setters  
}  
class HourlyEmployee {
```

```
    int eno;  
    String ename;  
    Int workedHours;  
    Double costPerHour;  
    // getters & setters  
}
```

⇒ In above pojo classes eno, ename is repeated properties in both pojo's. So instead of this write these two common properties in single pojo called Employee. And then extend this Employee to both pojo's.

```
class Employee{  
    int eno;  
    String ename;  
    // getters & setters  
}
```

```
class SalariedEmployee extends Employee {  
    Double salary;  
    // getters & setters  
}
```

```
class HourlyEmployee extends Employee {  
    Int workedHours;  
    Double costPerHour;  
    // getters & setters  
}
```

⇒ For the above two pojos there are three ways of database design.

1st way (Table Per concrete class)

HOURLY_EMPLOYEE
ENO ENAME WORKED_HOURS COST_PER_HOUR

SALARIED_EMPLOYEE
ENO ENAME SALARY

2nd way (table per class)

EMPLOYEE
ENO ENAME EMPLOYEE_DESC WORKED_HOURS COST_PER_HOUR SALARY

3rd way(table per subclass)

EMPLOYEE
ENO ENAME

SALARIED_EMPLOYEE
ENO SALARY

HOURLY_EMPLOYEE
ENO WORKED_HOURS COST_PER_HOUR

```
└─ table_per_concrete_class-hierarchy
    └─ src
        └─ com.sekharit.hibernate.config
            └─ hibernate.cfg.xml
        └─ com.sekharit.hibernate.dao
            └─ CreateDAO.java
            └─ DeleteDAO.java
            └─ RetrieveDAO.java
            └─ UpdateDAO.java
        └─ com.sekharit.hibernate.entity
            └─ Employee.java
            └─ HourlyEmployee.java
            └─ SalariedEmployee.java
        └─ com.sekharit.hibernate.mapping
            └─ Employee.hbm.xml
            └─ HourlyEmployee.hbm.xml
            └─ SalariedEmployee.hbm.xml
        └─ com.sekharit.hibernate.util
            └─ SessionUtil.java
    └─ JRE System Library
    └─ Referenced Libraries
    └─ Hibernate 4.x Libraries
```

Relation between the tables is like the following:

EMP	
ENO	NUMBER(10)
NAME	VARCHAR2(100)

HOUR_EMP	
ENO	NUMBER(10)
NAME	VARCHAR2(100)
WORKED_HOURS	NUMBER(10)
COST_PER_HOUR	FLOAT(10)

SAL_EMP	
ENO	NUMBER(10)
NAME	VARCHAR2(100)
ESAL	FLOAT(10)
BONUS	FLOAT(10)

Employee.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.Inheritance;
9. import javax.persistence.InheritanceType;
10. import javax.persistence.Table;
11.
12. import org.hibernate.annotations.GenericGenerator;
13.
14. @Entity
15. @Table(name = "EMP")
16. @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
17. public class Employee {
18.     @Id
19.     @GenericGenerator(name = "myGenerator", strategy = "increment")
20.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
21.     @Column(name = "NO")
22.     private int eno;
23.     @Column(name = "NAME")
24.     private String name;
25.
26.     //getters & setters
27. }
```

HourlyEmployee.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.AttributeOverride;
4. import javax.persistence.Column;
5. import javax.persistence.Entity;
6. import javax.persistence.Table;
7.
8. @Entity
9. @Table(name="HOUR_EMP")
10. @AttributeOverride(name = "name", column = @Column(name = "ENAME"))
11. public class HourlyEmployee extends Employee {
12.     @Column(name="WORKED_HOURS")
13.     private int workedHours;
14.     @Column(name="COST_PER_HOUR")
15.     private double costPerHour;
16.
17.     //getters & setters
18. }
```

SalariedEmployee.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.AttributeOverride;
4. import javax.persistence.AttributeOverrides;
5. import javax.persistence.Column;
6. import javax.persistence.Entity;
7. import javax.persistence.Table;
8.
9. @Entity
10. @Table(name = "SAL_EMP")
11. @AttributeOverrides( {
12.     @AttributeOverride(name = "eno", column = @Column(name = "EID")),
13.     @AttributeOverride(name = "name", column = @Column(name = "ENAME")) })
14. public class SalariedEmployee extends Employee {
15.     @Column(name = "SAL")
16.     private double salary;
17.     @Column(name = "BONUS")
18.     private double bonus;
19.
20.     //getters & setters
21. }
```

Employee.hbm.xml

```

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3.     "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4.     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5. <hibernate-mapping>
6.     <class name="com.sekharit.hibernate.entity.Employee" table="EMP">
7.         <id name="eno" column="ENO">
8.             <generator class="increment"></generator>
9.         </id>
10.        <property name="name" column="NAME"></property>
11.    </class>
12. </hibernate-mapping>
```

HourlyEmployee.hbm.xml

```

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3.     "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4.     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5. <hibernate-mapping>
6.     <class name="com.sekharit.hibernate.entity.HourlyEmployee" table="HOUR_EMP">
7.         <id name="eno" column="ENO">
8.             <generator class="increment"></generator>
9.         </id>
10.        <property name="name" column="NAME"></property>
11.        <property name="workedHours" column="WORKED_HOURS"></property>
12.        <property name="costPerHour" column="COST_PER_HOUR"></property>
13.    </class>
14. </hibernate-mapping>
```

SalariedEmployee.hbm.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3.         "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4.         "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5. <hibernate-mapping>
6.     <class name="com.sekharit.hibernate.entity.SalariedEmployee"
7.             table="SAL_EMP">
8.         <id name="eno" column="ENO">
9.             <generator class="increment"></generator>
10.        </id>
11.        <property name="name" column="NAME"></property>
12.        <property name="salary" column="ESAL"></property>
13.        <property name="bonus" column="BONUS"></property>
14.    </class>
15. </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.entity.HourlyEmployee;
7. import com.sekharit.hibernate.entity.SalariedEmployee;
8. import com.sekharit.hibernate.util.SessionUtil;
9.
10. public class CreateDAO {
11.     public static void main(String[] args) {
12.         Session session = SessionUtil.getSession();
13.         session.beginTransaction();
14.
15.         Employee employee = new Employee();
16.         employee.setName("somasekhar");
17.
18.         SalariedEmployee salEmp = new SalariedEmployee();
19.         salEmp.setName("sekhar");
20.         salEmp.setSalary(2500.0);
21.         salEmp.setBonus(450);
22.
23.         HourlyEmployee hourEmp = new HourlyEmployee();
24.         hourEmp.setName("sому");
25.         hourEmp.setWorkedHours(29);
26.         hourEmp.setCostPerHour(260.9);
27.
28.         session.save(employee);
29.         session.save(salEmp);
30.         session.save(hourEmp);
31.
32.         session.getTransaction().commit();
33.         SessionUtil.closeSession(session);
34.     }
35. }
```

OUTPUT TABLES:

EMP	SAL_EMP	HOUR_EMP
ENO NAME	ENO NAME ESAL BONUS	ENO NAME WORKED_HOURS COST_PER_HOUR
1 somasekhar	1 sekhar 2500 450	1 somu 29 260.9

RetrieveDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.entity.HourlyEmployee;
7. import com.sekharit.hibernate.entity.SalariedEmployee;
8. import com.sekharit.hibernate.util.SessionUtil;
9.
10. public class RetrieveDAO {
11.     public static void main(String[] args) {
12.         Session session = SessionUtil.getSession();
13.         session.beginTransaction();
14.
15.         Employee employee = (Employee) session.get(Employee.class, 1);
16.         sop("employee details...");
17.         sop("Eno : " + employee.getEno());
18.         sop("Name : " + employee.getName());
19.
20.         SalariedEmployee salEmp = (SalariedEmployee) session.get(
21.             SalariedEmployee.class, 2);
22.         sop("salaried employee details....");
23.         sop("Eno : " + salEmp.getEno());
24.         sop("Name : " + salEmp.getName());
25.         sop("Salary : " + salEmp.getSalary());
26.         sop("Bonus : " + salEmp.getBonus());
27.
28.         HourlyEmployee hourEmp = (HourlyEmployee) session.get(
29.             HourlyEmployee.class, 3);
30.         sop("Hourly Employee details...");
31.         sop("Eno : " + hourEmp.getEno());
32.         sop("Name : " + hourEmp.getName());
33.         sop("Worked Hours : " + hourEmp.getWorkedHours());
34.         sop("Cost Per Hour : " + hourEmp.getCostPerHour());
35.
36.         session.beginTransaction().commit();
37.         SessionUtil.closeSession(session);
38.     }
39.
40.     public static void sop(Object object) {
41.         System.out.println(object);
42.     }
43. }
44.
```

OUTPUT:

employee details...
Eno : 1

```
Name : somasekhar
```

```
salaried employee details....
```

```
Eno : 1
```

```
Name : sekhar
```

```
Salary : 2500.0
```

```
Bonus : 450.0
```

```
Hourly Employee details...
```

```
Eno : 1
```

```
Name : somu
```

```
Worked Hours : 29
```

```
Cost Per Hour : 260.9
```

UpdateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.entity.HourlyEmployee;
7. import com.sekharit.hibernate.entity.SalariedEmployee;
8. import com.sekharit.hibernate.util.SessionUtil;
9.
10. public class UpdateDAO {
11.     public static void main(String[] args) {
12.         Session session = SessionUtil.getSession();
13.         session.beginTransaction();
14.
15.         Employee employee = (Employee) session.get(Employee.class, 1);
16.         employee.setName("new sekhar");
17.
18.         SalariedEmployee salEmp = (SalariedEmployee) session.get(
19.             SalariedEmployee.class, 1);
20.         salEmp.setSalary(1400.9);
21.
22.         HourlyEmployee hourEmp = (HourlyEmployee) session.get(
23.             HourlyEmployee.class, 1);
24.         hourEmp.setWorkedHours(15);
25.
26.         session.getTransaction().commit();
27.         SessionUtil.closeSession(session);
28.     }
29.
30. }
```

OUTPUT TABLES:

EMP		HOUR_EMP			SAL_EMP				
ENO	NAME	ENO	NAME	WORKED_HOURS	COST_PER_HOUR	ENO	NAME	ESAL	BONUS
1	new sekhar	1	somu	15	260.9	1	sekhar	1400.9	450

DeleteDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
```

```

3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.entity.HourlyEmployee;
7. import com.sekharit.hibernate.entity.SalariedEmployee;
8. import com.sekharit.hibernate.util.SessionUtil;
9.
10. public class DeleteDAO {
11.     public static void main(String[] args) {
12.         Session session = SessionUtil.getSession();
13.         session.beginTransaction();
14.
15.         Employee emp = (Employee) session.get(Employee.class, 1);
16.
17.         SalariedEmployee salEmp = (SalariedEmployee) session.get(
18.             SalariedEmployee.class, 1);
19.
20.         HourlyEmployee hourEmp = (HourlyEmployee) session.get(
21.             HourlyEmployee.class, 1);
22.
23.         session.delete(emp);
24.         session.delete(salEmp);
25.         session.delete(hourEmp);
26.
27.         session.getTransaction().commit();
28.         SessionUtil.closeSession(session);
29.     }
30. }
31. }
```

OUTPUT TABLES:

EMP		HOUR_EMP			SAL_EMP				
ENO	NAME	ENO	NAME	WORKED_HOURS	COST_PER_HOUR	ENO	NAME	ESAL	BONUS
1	ABC	1	ABC	100	100	1	ABC	1000	100

```

    ↳ table_per_class-hierarchy
      ↳ src
        ↳ com.sekharit.hibernate.config
          ↳ hibernate.cfg.xml
        ↳ com.sekharit.hibernate.dao
          ↳ CreateDAO.java
          ↳ DeleteDAO.java
          ↳ RetrieveDAO.java
          ↳ UpdateDAO.java
        ↳ com.sekharit.hibernate.entity
          ↳ Employee.java
          ↳ HourlyEmployee.java
          ↳ SalariedEmployee.java
        ↳ com.sekharit.hibernate.mapping
          ↳ Employee.hbm.xml
        ↳ com.sekharit.hibernate.util
          ↳ SessionUtil.java
      ↳ JRE System Library
      ↳ Hibernate 4.x Libraries
      ↳ Referenced Libraries
  
```

Relation between the tables is like the following:

EMP	
ENO	NUMBER(10)
EMP_DESC	VARCHAR2(1020)
NAME	VARCHAR2(1020)
SAL	FLOAT(126)
BONUS	FLOAT(126)
WORKED_HOURS	NUMBER(10)
COST_PER_HOUR	FLOAT(126)

Employee.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.DiscriminatorColumn;
5. import javax.persistence.DiscriminatorType;
6. import javax.persistence.DiscriminatorValue;
7. import javax.persistence.Entity;
8. import javax.persistence.GeneratedValue;
9. import javax.persistence.GenerationType;
10. import javax.persistence.Id;
11. import javax.persistence.Inheritance;
12. import javax.persistence.InheritanceType;
13. import javax.persistence.Table;
14.
15. import org.hibernate.annotations.GenericGenerator;
16.
17. @Entity
18. @Table(name = "EMP")
19. @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
20. @DiscriminatorColumn(name = "EMP_DESC",discriminatorType=DiscriminatorType.STRING)
  
```

```
21. @DiscriminatorValue(value = "emp")
22. public class Employee {
23.     @Id
24.     @GenericGenerator(name = "myGenerator", strategy = "increment")
25.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
26.     @Column(name = "ENO")
27.     private int eno;
28.     @Column(name = "NAME")
29.     private String name;
30.
31.     //getters & setters
32. }
```

HourlyEmployee.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.DiscriminatorValue;
5. import javax.persistence.Entity;
6. import javax.persistence.Table;
7.
8. @Entity
9. @Table(name="EMP")
10. @DiscriminatorValue("hourlyEmp")
11. public class HourlyEmployee extends Employee {
12.     @Column(name="WORKED_HOURS")
13.     private int workedHours;
14.     @Column(name="COST_PER_HOUR")
15.     private double costPerHour;
16.
17.     //getters & setters
18. }
```

SalariedEmployee.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.DiscriminatorValue;
5. import javax.persistence.Entity;
6. import javax.persistence.Table;
7.
8. @Entity
9. @Table(name = "EMPLOYEE")
10. @DiscriminatorValue("salariedEmp")
11. public class SalariedEmployee extends Employee {
12.     @Column(name = "SAL")
13.     private double salary;
14.     @Column(name = "BONUS")
15.     private double bonus;
19.     //getters & setters
16. }
```

Employee.hbm.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
```

```
3.         "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4.         "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
5. <hibernate-mapping>
6.   <class name="com.sekharit.hibernate.entity.Employee" table="EMP"
7.     discriminator-value="emp">
8.     <id name="eno" column="ENO">
9.       <generator class="increment"></generator>
10.    </id>
11.    <discriminator column="EMP_DESC"></discriminator>
12.    <!-- descriminator tag must write immediately after id tag -->
13.    <property name="name" column="NAME"></property>
14.    <subclass name="com.sekharit.hibernate.entity.SalariedEmployee"
15.      discriminator-value="salariedEmp">
16.        <property name="salary" column="SAL"></property>
17.        <property name="bonus" column="BONUS"></property>
18.      </subclass>
19.      <subclass name="com.sekharit.hibernate.entity.HourlyEmployee"
20.        discriminator-value="hourlyEmp">
21.          <property name="workedHours" column="WORKED_HOURS"></property>
22.          <property name="costPerHour" column="COST_PER_HOUR"></property>
23.        </subclass>
24.      </class>
25. </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.entity.HourlyEmployee;
7. import com.sekharit.hibernate.entity.SalariedEmployee;
8. import com.sekharit.hibernate.util.SessionUtil;
9.
10. public class CreateDAO {
11.     public static void main(String[] args) {
12.         Session session = SessionUtil.getSession();
13.         session.beginTransaction();
14.
15.         Employee employee = new Employee();
16.         employee.setName("somu");
17.
18.         SalariedEmployee salEmp = new SalariedEmployee();
19.         salEmp.setName("sekhar");
20.         salEmp.setSalary(25000);
21.         salEmp.setBonus(4200);
22.
23.         HourlyEmployee hourEmp = new HourlyEmployee();
24.         hourEmp.setName("somasekhar");
25.         hourEmp.setWorkedHours(56);
26.         hourEmp.setCostPerHour(278.9);
27.
28.         session.save(employee);
29.         session.save(salEmp);
```

```

30.         session.save(hourEmp);
31.
32.         session.getTransaction().commit();
33.         SessionUtil.closeSession(session);
34.     }
35. }
```

OUTPUT TABLES:**EMP**

EMP_DESC	ENO	NAME	COST_PER_HOUR	WORKED_HOURS	BONUS	SAL
emp	1	somu				
salariedEmp	2	sekhar			4200	25000
hourlyEmp	3	somasekhar	278.9		56	

RetrieveDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.entity.HourlyEmployee;
7. import com.sekharit.hibernate.entity.SalariedEmployee;
8. import com.sekharit.hibernate.util.SessionUtil;
9.
10. public class RetrieveDAO {
11.     public static void main(String[] args) {
12.         Session session = SessionUtil.getSession();
13.         session.beginTransaction();
14.
15.         // Employee employee = (Employee) session.get(Employee.class, 1);
16.         // Employee employee = (Employee) session.get(Employee.class, 2);
17.         Employee employee = (Employee) session.get(Employee.class, 3);
18.
19.         if (employee instanceof HourlyEmployee) {
20.             HourlyEmployee hourempl = (HourlyEmployee) employee;
21.             sop("Hourly Employee details... ");
22.             sop("Eno : " + hourempl.getEno());
23.             sop("Name : " + hourempl.getName());
24.             sop("Worked Hours : " + hourempl.getWorkedHours());
25.             sop("Cost Per Hour : " + hourempl.getCostPerHour());
26.         } else if (employee instanceof SalariedEmployee) {
27.             SalariedEmployee salemp = (SalariedEmployee) employee;
28.             sop("salaried employee details... ");
29.             sop("Eno : " + salemp.getEno());
30.             sop("Name : " + salemp.getName());
31.             sop("Salary : " + salemp.getSalary());
32.             sop("Bonus : " + salemp.getBonus());
33.         } else {
34.             sop("Employee details... ");
35.             sop("Eno : " + employee.getEno());
36.             sop("Name : " + employee.getName());
37.         }
38.     }
}
```

```
39.         session.getTransaction().commit();
40.         SessionUtil.closeSession(session);
41.     }
42.
43.     public static void sop(Object object) {
44.         System.out.println(object);
45.     }
46. }
```

OUTPUT:

Hourly Employee details...

Eno : 3
Name : somasekhar
Worked Hours : 56
Cost Per Hour : 278.9

UpdateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.entity.HourlyEmployee;
7. import com.sekharit.hibernate.entity.SalariedEmployee;
8. import com.sekharit.hibernate.util.SessionUtil;
9.
10. public class UpdateDAO {
11.     public static void main(String[] args) {
12.         Session session = SessionUtil.getSession();
13.         session.beginTransaction();
14.
15.         // Employee employee = (Employee)session.get(Employee.class, 1);
16.         // Employee employee = (Employee)session.get(Employee.class, 2);
17.         Employee employee = (Employee) session.get(Employee.class, 3);
18.
19.         if (employee instanceof HourlyEmployee) {
20.             HourlyEmployee hourEmp = (HourlyEmployee) employee;
21.             hourEmp.setWorkedHours(98);
22.         } else if (employee instanceof SalariedEmployee) {
23.             SalariedEmployee salEmp = (SalariedEmployee) employee;
24.             salEmp.setSalary(2590.6);
25.         } else {
26.             employee.setName("yerragudi");
27.         }
28.
29.         session.getTransaction().commit();
30.         SessionUtil.closeSession(session);
31.     }
32. }
33. }
```

OUTPUT TABLE:

EMP_DESC	ENO	NAME	COST_PER_HOUR	WORKED_HOURS	BONUS	SAL
emp	1	somu				
salariedEmp	2	sekhar			4200	25000
hourlyEmp	3	somasekhar	278.9	98		

DeleteDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.util.SessionUtil;
7.
8. public class DeleteDAO {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         // Employee employee=(Employee)session.get(Employee.class, 1);
14.         // Employee employee=(Employee)session.get(Employee.class, 2);
15.         Employee employee = (Employee) session.get(Employee.class, 3);
16.         session.delete(employee);
17.
18.         session.getTransaction().commit();
19.         SessionUtil.closeSession(session);
20.     }
21.
22. }
```

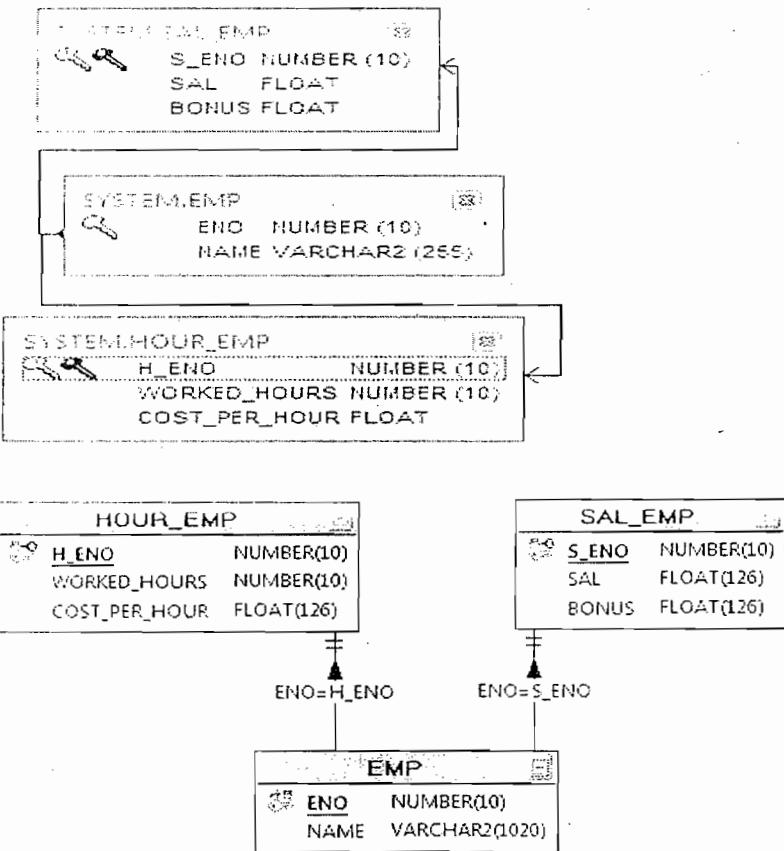
OUTPUT TABLE:

EMP_DESC	ENO	NAME	COST_PER_HOUR	WORKED_HOURS	BONUS	SAL
emp	1	somu				
salariedEmp	2	sekhar			4200	25000

```

.
.
.
└── table_per_sub_class-hierarchy
    └── src
        └── com.sekharit.hibernate.config
            └── hibernate.cfg.xml
        └── com.sekharit.hibernate.dao
            ├── CreateDAO.java
            ├── DeleteDAO.java
            ├── RetrieveDAO.java
            └── UpdateDAO.java
        └── com.sekharit.hibernate.entity
            ├── Employee.java
            ├── HourlyEmployee.java
            └── SalariedEmployee.java
    └── com.sekharit.hibernate.mapping
        └── Employee.hbm.xml
    └── com.sekharit.hibernate.util
        └── SessionUtil.java
└── JRE System Library [jre6]
└── Hibernate 4.x Libraries
└── Referenced Libraries

```



Employee.java

```

1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.GeneratedValue;

```

```
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.Inheritance;
9. import javax.persistence.InheritanceType;
10. import javax.persistence.Table;
11.
12. import org.hibernate.annotations.GenericGenerator;
13.
14. @Entity
15. @Table(name = "EMP")
16. @Inheritance(strategy = InheritanceType.JOINED)
17. public class Employee {
18.     @Id
19.         @GenericGenerator(name = "myGenerator", strategy = "increment")
20.         @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
21.         @Column(name = "ENO")
22.         private int eno;
23.
24.         @Column(name = "NAME")
25.         private String name;
26.
27.     //getters & setters
28. }
```

HourlyEmployee.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.PrimaryKeyJoinColumn;
6. import javax.persistence.Table;
7.
8. @Entity
9. @Table(name = "HOUR_EMP")
10. @PrimaryKeyJoinColumn(name="H_ENO")
11. public class HourlyEmployee extends Employee {
12.     @Column(name = "WORKED_HOURS")
13.     private int workedHours;
14.     @Column(name = "COST_PER_HOUR")
15.     private double costPerHour;
16.
17.     //getters & setters
18. }
```

SalariedEmployee.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.PrimaryKeyJoinColumn;
6. import javax.persistence.Table;
7.
8. @Entity
9. @Table(name = "SAL_EMP")
10. @PrimaryKeyJoinColumn(name = "S_ENO")
```

```
11. public class SalariedEmployee extends Employee {  
12.     @Column(name = "SAL")  
13.     private double salary;  
14.     @Column(name = "BONUS")  
15.     private double bonus;  
16.  
19.         //getters & setters  
17. }
```

Employee.hbm.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>  
2. <!DOCTYPE hibernate-mapping PUBLIC  
3.         "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
4.         "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  
5. <hibernate-mapping>  
6.     <class name="com.sekharit.hibernate.entity.Employee" table="EMP">  
7.         <id name="eno" column="ENO">  
8.             <generator class="increment"></generator>  
9.         </id>  
10.        <property name="name" column="NAME"></property>  
11.        <joined-subclass name="com.sekharit.hibernate.entity.SalariedEmployee"  
12.            table="SAL_EMP">  
13.            <key column="S_ENO"></key>  
14.            <property name="salary" column="SAL"></property>  
15.            <property name="bonus" column="BONUS"></property>  
16.        </joined-subclass>  
17.        <joined-subclass name="com.sekharit.hibernate.entity.HourlyEmployee"  
18.            table="HOUR_EMP">  
19.            <key column="H_ENO"></key>  
20.            <property name="workedHours" column="WORKED_HOURS"></property>  
21.            <property name="costPerHour" column="COST_PER_HOUR"></property>  
22.        </joined-subclass>  
23.    </class>  
24. </hibernate-mapping>
```

CreateDAO.java

```
1. package com.sekharit.hibernate.dao;  
2.  
3. import org.hibernate.Session;  
4.  
5. import com.sekharit.hibernate.entity.Employee;  
6. import com.sekharit.hibernate.entity.HourlyEmployee;  
7. import com.sekharit.hibernate.entity.SalariedEmployee;  
8. import com.sekharit.hibernate.util.SessionUtil;  
9.  
10. public class CreateDAO {  
11.     public static void main(String[] args) {  
12.         Session session = SessionUtil.getSession();  
13.         session.beginTransaction();  
14.  
15.         Employee employee = new Employee();  
16.         employee.setName("somasekhar");  
17.  
18.         SalariedEmployee salEmp = new SalariedEmployee();
```

```

19.         salEmp.setName("sekhar");
20.         salEmp.setSalary(25000);
21.         salEmp.setBonus(659);
22.
23.         HourlyEmployee hourEmp = new HourlyEmployee();
24.         hourEmp.setName("somu");
25.         hourEmp.setWorkedHours(56);
26.         hourEmp.setCostPerHour(278.9);
27.
28.         session.save(employee);
29.         session.save(salEmp);
30.         session.save(hourEmp);
31.
32.         session.getTransaction().commit();
33.         SessionUtil.closeSession(session);
34.     }
35. }
```

Output Tables:

EMP		SAL_EMP			HOUR_EMP		
ENO	NAME	S_ENO	SAL	BONUS	H_ENO	WORKED_HOURS	COST_PER_HOUR
1	somasekhar	2	25000	659	3	56	278.9
2	sekhar						
3	somu						

RetrieveDAO.java

```

1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.entity.HourlyEmployee;
7. import com.sekharit.hibernate.entity.SalariedEmployee;
8. import com.sekharit.hibernate.util.SessionUtil;
9.
10. public class RetrieveDAO {
11.     public static void main(String[] args) {
12.         Session session = SessionUtil.getSession();
13.         session.beginTransaction();
14.
15.         // Employee employee = (Employee) session.get(Employee.class, 1);
16.         // Employee employee = (Employee) session.get(Employee.class, 2);
17.         Employee employee = (Employee) session.get(Employee.class, 3);
18.
19.         if (employee instanceof HourlyEmployee) {
20.             HourlyEmployee hourEmp = (HourlyEmployee) employee;
21.             sop("Hourly Employee details... ");
22.             sop("Eno : " + hourEmp.getEno());
23.             sop("Name : " + hourEmp.getName());
24.             sop("Worked Hours : " + hourEmp.getWorkedHours());
25.             sop("Cost Per Hour : " + hourEmp.getCostPerHour());
26.         } else if (employee instanceof SalariedEmployee) {
27.             SalariedEmployee salEmp = (SalariedEmployee) employee;
```

```
28.             sop("salaried employee details....");
29.             sop("Eno : " + salEmp.getEno());
30.             sop("Name : " + salEmp.getName());
31.             sop("Salary : " + salEmp.getSalary());
32.             sop("Bonus : " + salEmp.getBonus());
33.         } else {
34.             sop("employee details...");
35.             sop("Eno : " + employee.getEno());
36.             sop("Name : " + employee.getName());
37.         }
38.     }
39.
40.     session.getTransaction().commit();
41.     SessionUtil.closeSession(session);
42. }
43.
44. public static void sop(Object object) {
45.     System.out.println(object);
46. }
47. }
```

Output:

```
Hourly Employee details...
Eno : 3
Name : somu
Worked Hours : 56
Cost Per Hour : 278.9
```

UpdateDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.entity.HourlyEmployee;
7. import com.sekharit.hibernate.entity.SalariedEmployee;
8. import com.sekharit.hibernate.util.SessionUtil;
9.
10. public class UpdateDAO {
11.     public static void main(String[] args) {
12.         Session session = SessionUtil.getSession();
13.         session.beginTransaction();
14.
15.         // Employee employee=(Employee)session.get(Employee.class, 1);
16.         // Employee employee=(Employee)session.get(Employee.class, 2);
17.         Employee employee = (Employee) session.get(Employee.class, 3);
18.
19.         if (employee instanceof HourlyEmployee) {
20.             HourlyEmployee hourEmp = (HourlyEmployee) employee;
21.             hourEmp.setWorkedHours(98);
22.         } else if (employee instanceof SalariedEmployee) {
23.             SalariedEmployee salEmp = (SalariedEmployee) employee;
24.             salEmp.setSalary(2590.6);
25.         } else {
```

Hibernate-'is-a' relationship mismatch

By Mr. SekharReddy

```
26.             employee.setName("new Sekhar");
27.         }
28.
29.         session.getTransaction().commit();
30.         SessionUtil.closeSession(session);
31.     }
32.
33. }
```

Output Tables:

EMP		SAL_EMP			HOUR_EMP		
ENO	NAME	S_ENO	SAL	BONUS	H_ENO	WORKED_HOURS	COST_PER_HOUR
1	somasekhar	2	25000	659	3	98	278.9
2	sekhar						
3	samu						

DeleteDAO.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4.
5. import com.sekharit.hibernate.entity.Employee;
6. import com.sekharit.hibernate.util.SessionUtil;
7.
8. public class DeleteDAO {
9.     public static void main(String[] args) {
10.         Session session = SessionUtil.getSession();
11.         session.beginTransaction();
12.
13.         // Employee employee=(Employee)session.get(Employee.class, 1);
14.         // Employee employee=(Employee)session.get(Employee.class, 2);
15.         Employee employee = (Employee) session.get(Employee.class, 3);
16.         session.delete(employee);
17.
18.         session.getTransaction().commit();
19.         SessionUtil.closeSession(session);
20.     }
21.
22. }
```

OUTPUT TABLES:

EMP		SAL_EMP			HOUR_EMP		
ENO	NAME	S_ENO	SAL	BONUS	H_ENO	WORKED_HOURS	COST_PER_HOUR
1	somasekhar	2	25000	659	3	98	278.9
2	sekhar						

Caching

- ⇒ There are two types of caching techniques in hibernate.
 - 1st level caching – Session
 - 2nd level cahing – SessionFactory.

1st level caching:

- ⇒ To implement 1st level caching there is no need of any explicit configuration. Because it is default caching.

Exmple : SessionLevelCahce in EHCache

2nd level caching:

- ⇒ There are so many 2nd level caching implementations.

Cache	Read-only	Nonstrict read/write	Read/write	transactional
1. EHCache	YES	YES	YES	NO
2. OSCache	YES	YES	YES	NO
3. SwarmCache	YES	YES	NO	NO
4. JbossTreeCache	YES	NO	NO	YES

1. EHCache:

- ⇒ EHCache is first, lightweight and easy-to-use in process cache. It supports read-only and read/write caching and memory and disk-based caching. However it doesn't support clustering.

2. OSCache:

- ⇒ OSCache is another open-source caching solution. It is part of a lager package, which also provides caching functionalities for jsp pages or arbitary objects. It is a powerful and flexible package. Which is like EHCache, supports read-only and read/write caching and memory and disk based caching. It is also provides basic support for clustering via either java groups or JMS.

3. SwarmCache :

- ⇒ It is a simple cluster based caching solution based on java groups. It supports readonly and non strict read/write caching (the next section explain this term). This type of cache is appropriate for applications that typically have many more read operations then write operations.

4. Jboss TreeCache:

- ⇒ It is powerful replicated(synchronous or asynchronous) and transactional cache. Use this section if you really need a true transaction-capable caching architecture.

Caching strategies

Read-only:

- ⇒ This strategy is useful for data that is read frequently but never updated this is by far the simplest and best performing cache strategy.

Read/write:

- ⇒ This cache may be appropriate if your data needs to be updated. They carry more overhead than read-only caches. In non JTA environments each transaction should be completed when session.close() or session.disconnect() is called.

Non strict read/write:

- ⇒ This strategy doesn't guarantee that two transactions can't simultaneously modify the same data. Therefore it may be most appropriate for data that is read often but only occasionally modified.

Transactional :

- ⇒ This is a fully transactional cache that may be used only in a JTA environment.

Examples of Constant / Lookup tables : COUNTRIES, STATES, ZIPCODES.....etc.

Steps to configure EHCache:

Step 1 : in the configuration file configure provider class and 2nd level cache and query cache entries.

```
<!--cache -->
<property name="hibernate.cache.use_query_cache">true</property>
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</property>
```

Step 2: configure ehcache.xml file. In this file we need to configure default cache strategies means how many times it has to hit the Database, max how many objects are required...etc.

```
<defaultCache
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true"
```

/>

The following attributes are required for defaultCache:

maxInMemory - Sets the maximum number of objects that will be created in memory

eternal - Sets whether elements are eternal. If eternal, timeouts are ignored and the element is never expired.

timeToIdleSeconds - Sets the time to idle for an element before it expires. Is only used if the element is not eternal. Idle time is now - last accessed time

timeToLiveSeconds - Sets the time to live for an element before it expires. Is only used if the element is not eternal. TTL is now - creation time

overflowToDisk - Sets whether elements can overflow to disk when the in-memory cache has reached the maxInMemory limit.

- ⇒ If you want to apply separate configuration for a table, then we need to configure the pojo name and setting in the ehcache.xml.

```
<cache name="com.sekharit.entity.Account"  
      maxElementsInMemory="10000"  
      eternal="false"  
      timeToIdleSeconds="300"  
      timeToLiveSeconds="600"  
      overflowToDisk="true"/>  
/>
```

Q.) How to apply configured EHCache to intended pojo's.

- ⇒ In the corresponding HBM file give the below entry.
 - <cache usage="read-only" />

Example : EHCacheProject

OScache:

Steps to configure OScache

Step 1: In cfg file provide OScache related entries.

```
<property name="hibernate.cache.use_query_cache">true</property>
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.provider_class">
com.opensymphony.oscache.hibernate.OSCacheProvider</property>
<property name="com.opensymphony.oscache.configurationResourceName">
oscache.properties</property>
```

Step 2: configure osCahce.properties.

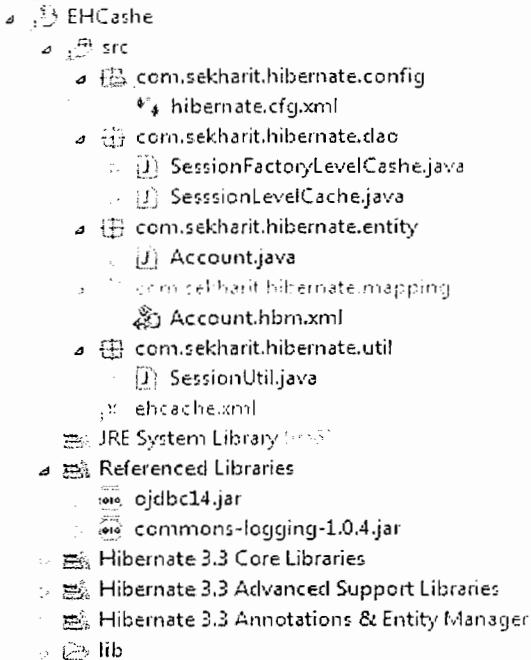
refresh.period=60000

cache.capacity=1000

Note : model ehcache.xml, oscache.properties files are available in hibernate-distribution-x.x.x-dist.jar. By extracting this file we get model files.

Example : OSCacheProject

EHCache



hibernate.cfg.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
```

```
3.           "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.           "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5. <hibernate-configuration>
6.
7.   <session-factory>
8.     <property name="dialect">
9.       org.hibernate.dialect.OracleDialect
10.      </property>
11.      <property name="connection.url">
12.        jdbc:oracle:thin:@localhost:1521:XE
13.      </property>
14.      <property name="connection.username">system</property>
15.      <property name="connection.password">tiger</property>
16.      <property name="connection.driver_class">
17.        oracle.jdbc.driver.OracleDriver
18.      </property>
19.      <property name="myeclipse.connection.profile">
20.        oracledriver
21.      </property>
22.      <property name="show_sql">true</property>
23.      <property name="hibernate.cache.use_query_cache">true</property>
24.      <property name="hibernate.cache.use_second_level_cache">
25.        true
26.      </property>
27.      <property name="hibernate.cache.provider_class">
28.        org.hibernate.cache.EhCacheProvider
29.      </property>
30.      <mapping resource="com/sekharit/hibernate/mapping/Account.hbm.xml" />
31.    </session-factory>
32.
33. </hibernate-configuration>
```

Account.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
6.     <cache usage="read-only" />
7.     <id name="accno" column="ACCNO"></id>
8.     <property name="name" column="NAME"></property>
9.     <property name="balance" column="BALANCE"></property>
10.   </class>
11. </hibernate-mapping>
```

Account.java

```
1. package com.sekharit.hibernate.entity;
2. import javax.persistence.Column;
3. import javax.persistence.Entity;
4. import javax.persistence.Id;
5. import javax.persistence.Table;
6. import org.hibernate.annotations.Cache;
7. import org.hibernate.annotations.CacheConcurrencyStrategy;
8.
9. @Entity
```

```
10. @Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
11. @Table(name = "ACCOUNT")
12. public class Account {
13.     @Id
14.     @Column(name = "ACCNO")
15.     private int accno;
16.     @Column(name = "NAME")
17.     private String name;
18.     @Column(name = "BALANCE")
19.     private float balance;
```

SessionLevelCache.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class SessionLevelCache {
6.
7.     public static void main(String[] args) {
8.         Session session = SessionUtil.getSession();
9.
10.        Account account = (Account) session.get(Account.class, 1);
11.        System.out.println("Name : " + account.getName());
12.        System.out.println("Balance : " + account.getBalance());
13.
14.        account = (Account) session.get(Account.class, 1);
15.        System.out.println("Name : " + account.getName());
16.        System.out.println("Balance : " + account.getBalance());
17.
18.    }
19.
20. }
```

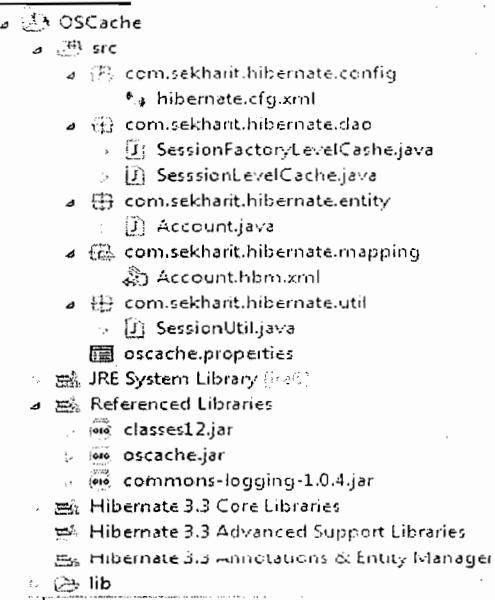
SessionFactoryLevelCashe.java

```
1. package com.sekharit.hibernate.dao;
2. import org.hibernate.Session;
3. import com.sekharit.hibernate.entity.Account;
4. import com.sekharit.hibernate.util.SessionUtil;
5. public class SessionFactoryLevelCashe {
6.
7.     public static void main(String[] args) {
8.         Session session1 = SessionUtil.getSession();
9.         Session session2 = SessionUtil.getSession();
10.
11.        Account account = (Account) session1.get(Account.class, 1);
12.        System.out.println("Name : " + account.getName());
13.        System.out.println("Balance : " + account.getBalance());
14.
15.        account = (Account) session2.get(Account.class, 1);
16.        System.out.println("Name : " + account.getName());
17.        System.out.println("Balance : " + account.getBalance());
18.
19.
20. }
```

ehcache.xml

```
1. <ehcache>
2.
3. <defaultCache
4.   maxElementsInMemory="10000"
5.   eternal="false"
6.   timeToIdleSeconds="120"
7.   timeToLiveSeconds="120"
8.   overflowToDisk="true"      />
9.
10. <cache
11.   name="com.nit.model.Account"
12.   maxElementsInMemory="10000"
13.   eternal="false"
14.   timeToIdleSeconds="300"
15.   timeToLiveSeconds="600"
16.   overflowToDisk="true"      />
17.
18. </ehcache>
```

OSCache



hibernate.cfg.xml

```
1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3.   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.   "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5.
6. <hibernate-configuration>
7.
8.   <session-factory>
9.     <property name="dialect">
10.       org.hibernate.dialect.OracleDialect
11.     </property>
12.     <property name="connection.url">
13.       jdbc:oracle:thin:@localhost:1521:XE
```

```
14.      </property>
15.          <property name="connection.username">system</property>
16.          <property name="connection.password">tiger</property>
17.          <property name="connection.driver_class">
18.              oracle.jdbc.driver.OracleDriver
19.          </property>
20.          <property name="myeclipse.connection.profile">
21.              oracledriver
22.          </property>
23.          <property name="show_sql">true</property>
24.          <property name="hibernate.cache.use_query_cache">true</property>
25.          <property name="hibernate.cache.use_second_level_cache">
26.              true
27.          </property>
28.          <property name="hibernate.cache.provider_class">
29.              com.opensymphony.oscache.hibernate.OSCacheProvider
30.          </property>
31.          <property name="com.opensymphony.oscache.configurationResourceName">
32.              oscache.properties
33.          </property>
34.          <mapping class="com.sekharit.hibernate.entity.Account" />
35.      </session-factory>
36.
37.  </hibernate-configuration>
```

Account.hbm.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3.  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4.  <hibernate-mapping>
5.      <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
6.          <cache usage="read-only" />
7.          <id name="accno" column="ACCNO"></id>
8.          <property name="name" column="NAME"></property>
9.          <property name="balance" column="BALANCE"></property>
10.         </class>
11.     </hibernate-mapping>
```

Account.java

== SAME AS ABOVE ==

SessionFactoryLevelCache.java

== SAME AS ABOVE ==

SessionFactoryLevelCashe.java

== SAME AS ABOVE ==

oscache.properties

```
1. refresh.period=60000
2. cache.capacity=1000
```

Version and Timestamp

- ⇒ Version is used to have updation count to table record value.
- ⇒ Timestamp is used to have time when exactly record was updated finally.
- ⇒ Lock() method locks total table records
- ⇒ Lock tables (ACCOUNT_LOCK) lock row (if more number of rows there it is not advisable)
- ⇒ Versioning and Timestamp are used to lock automatically.

Version

ACCOUNT			
ACCNO	NAME	BALANCE	VERSION
1009	Ramesh tendulkar	8900	1

⇒ Version column can have any name. That should be configured in HBM file.

Example :

VersionProject : VersionDemo1

- ⇒ <version> tag must be immediately followed by <id> tag.
- ⇒ While storing account object into database we no need to set version, because version value given by hibernate.
- ⇒ Version start from 0.

Testing:

- ⇒ Run VersionDemo2 in Debug mode.
- ⇒ Run VersionDemo3.
- ⇒ Then continue VersionDemo2 execution.

org.hibernate.StaleObjectStateException: Row was updated or deleted by another transaction (or unsaved-value mapping was incorrect): [com.nit.model.Account#1009]

- ⇒ It is because while inserting or updating record into the database it will check for version value. Version value should be same to the value when it retrieved from the database otherwise it throws Exception.
- ⇒ Logic just like this.

Account account=session.get(Account.class, 1004);

Say for example now version is (6);

While updating record

Again get version from the database

If it is not equal to '6' then it throws exception.

- ⇒ Remove <version> tag and do testing again you won't get Exception again.

Timestamp:

- ⇒ Timestamp will work same like version. We should have extra column which of type, which can hold time. The name of the column can be any name. Whenever we insert record into the database it stores the current time into timestamp column. And whenever the record gets updated again it updates the timestamp column with latest time.

Example: TimestampProject

- ⇒ Test same as VersionProject. But like VersionProject it won't throw Exception, But it won't update the record.

```
1. VersionProject
   2. src
      3. com.sekharit.hibernate.config
         4. hibernate.cfg.xml
      5. com.sekharit.hibernate.dao
         6. VersionDemo1.java
         7. VersionDemo2.java
         8. VersionDemo3.java
      9. com.sekharit.hibernate.entity
         10. Account.java
     11. com.sekharit.hibernate.mapping
         12. Account.hbm.xml
     13. com.sekharit.hibernate.util
         14. SessionUtil.java
  15. JRE System Library [JDK]
  16. Hibernate 4.x Libraries
  17. Referenced Libraries
```

Account.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
4. <hibernate-mapping>
5.   <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
6.     <id name="accno" column="ACCNO">
7.       <generator class="increment"></generator>
8.     </id>
9.     <version name="version" column="VERSION"></version>
10.    <property name="name" column="NAME"></property>
11.    <property name="balance" column="BALANCE"></property>
12.  </class>
13. </hibernate-mapping>
```

Account.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.GeneratedValue;
6. import javax.persistence.GenerationType;
7. import javax.persistence.Id;
8. import javax.persistence.Table;
9. import javax.persistence.Version;
10.
11. import org.hibernate.annotations.GenericGenerator;
12.
13. @Entity
14. @Table(name = "ACCOUNT")
15. public class Account {
16.
17.   @Id
18.   @GenericGenerator(name = "myGenerator", strategy = "increment")
19.   @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
20.   @Column(name = "ACCNO")
21.   private int accno;
```

```
22.     @Column(name = "NAME")
23.     private String name;
24.     @Column(name = "BALANCE")
25.     private float balance;
26.     @Version
27.     @Column(name = "VERSION")
28.     private int version;
29. // getters & setters
30. }
```

VersionDemo1.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4. import org.hibernate.Transaction;
5.
6. import com.sekharit.hibernate.entity.Account;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class VersionDemol {
10.
11.     public static void main(String[] args) {
12.
13.         Session session = SessionUtil.getSession();
14.         Transaction transaction = session.beginTransaction();
15.
16.         Account account = new Account();
17.         account.setName("sekhar");
18.         account.setBalance(900);
19.
20.         session.save(account);
21.
22.         System.out.println("Success");
23.         transaction.commit();
24.     }
25.
26. }
```

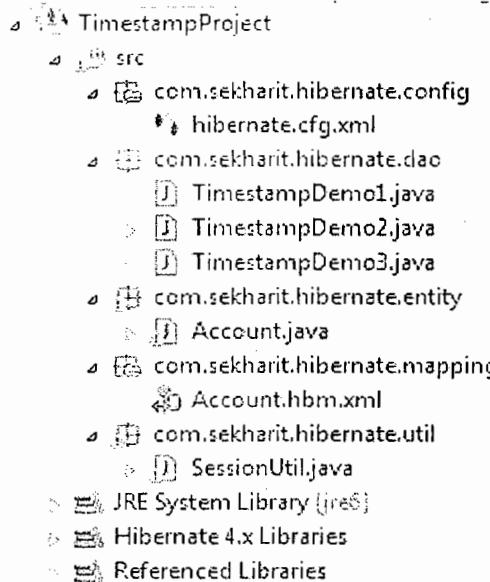
VersionDemo2.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4. import org.hibernate.Transaction;
5.
6. import com.sekharit.hibernate.entity.Account;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class VersionDemo2 {
10.
11.     public static void main(String[] args) {
12.
13.         Session session = SessionUtil.getSession();
14.         Transaction transaction = session.beginTransaction();
15.         Account account = (Account) session.get(Account.class, 1);
16.         System.out.println("...Break.point...And run VersionDemo3,
```

```
17.                     then come back and continue this prog execution");
18.             account.setBalance(9988.0f);
19.             transaction.commit();
20.         }
21.
22.     }
```

VersionDemo3.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4. import org.hibernate.Transaction;
5.
6. import com.sekharit.hibernate.entity.Account;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class VersionDemo3 {
10.
11.     public static void main(String[] args) {
12.
13.         Session session = SessionUtil.getSession();
14.         Transaction transaction = session.beginTransaction();
15.
16.         Account account = (Account) session.get(Account.class, 3);
17.         account.setName("somu");
18.
19.         transaction.commit();
20.
21.         System.out.println("Success");
22.
23.     }
24.
25. }
```



Account.hbm.xml

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

```
2. <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >
3. <hibernate-mapping>
4.   <class name="com.sekharit.hibernate.entity.Account" table="ACCOUNT">
5.     <id name="accno" column="ACCNO">
6.       <generator class="increment"></generator>
7.     </id>
8.     <timestamp name="timestamp" column="ACCOUNT_TIMESTAMP"></timestamp>
9.     <property name="name" column="NAME"></property>
10.    <property name="balance" column="BALANCE"></property>
11.  </class>
12. </hibernate-mapping>
```

Account.java

```
1. package com.sekharit.hibernate.entity;
2.
3. import java.util.Date;
4.
5. import javax.persistence.Column;
6. import javax.persistence.Entity;
7. import javax.persistence.GeneratedValue;
8. import javax.persistence.GenerationType;
9. import javax.persistence.Id;
10. import javax.persistence.Table;
11. import javax.persistence.Temporal;
12. import javax.persistence.TemporalType;
13.
14. import org.hibernate.annotations.GenericGenerator;
15.
16.
17. @Entity
18. @Table(name = "ACCOUNT")
19. public class Account {
20.
21.     @Id
22.     @GenericGenerator(name = "myGenerator", strategy = "increment")
23.     @GeneratedValue(strategy = GenerationType.AUTO, generator = "myGenerator")
24.     @Column(name = "ACCNO")
25.     private int accno;
26.     @Column(name = "NAME")
27.     private String name;
28.     @Column(name = "BALANCE")
29.     private float balance;
30.     @Temporal(TemporalType.TIMESTAMP)
31.     @Column(name = "ACCOUNT_TIMESTAMP")
32.     private Date timestamp;
33.     // getters & setters
34. }
```

TimestampDemo1.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4. import org.hibernate.Transaction;
5.
```

```
6. import com.sekharit.hibernate.entity.Account;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class TimestampDemo1 {
10.
11.     public static void main(String[] args) {
12.
13.         Session session = SessionUtil.getSession();
14.         Transaction transaction = session.beginTransaction();
15.
16.         Account account = new Account();
17.         account.setName("somasekhar");
18.         account.setBalance(8900);
19.
20.         session.save(account);
21.
22.         System.out.println("Success" );
23.         transaction.commit();
24.     }
25.
26. }
```

TimestampDemo2.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4. import org.hibernate.Transaction;
5.
6. import com.sekharit.hibernate.entity.Account;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class TimestampDemo2 {
10.
11.     public static void main(String[] args) {
12.
13.         Session session = SessionUtil.getSession();
14.         Transaction transaction = session.beginTransaction();
15.
16.         Account account = (Account) session.get(Account.class, 1);
17.         account.setName("kesavareddy");
18.
19.         System.out.println(..Break point..);
20.
21.         transaction.commit();
22.         System.out.println("Success");
23.     }
24.
25. }
```

TimestampDemo3.java

```
1. package com.sekharit.hibernate.dao;
2.
3. import org.hibernate.Session;
4. import org.hibernate.Transaction;
5.
```

```
6. import com.sekharit.hibernate.entity.Account;
7. import com.sekharit.hibernate.util.SessionUtil;
8.
9. public class TimestampDemo3 {
10.
11.     public static void main(String[] args) {
12.
13.         Session session = SessionUtil.getSession();
14.         Transaction transaction = session.beginTransaction();
15.
16.         Account account = (Account) session.get(Account.class, 1);
17.         account.setBalance(9567.0f);
18.
19.         System.out.println("Success");
20.         transaction.commit();
21.
22.     }
23.
24. }
```

inverse = “true” example and explanation

Always put inverse=”true” in your collection variable ?

There are many Hibernate articles try to explain the “inverse” with many Hibernate “official” jargon, which is very hard to understand (at least to me). In few articles, they even suggested that just forget about what is “inverse”, and always put inverse=”true” in the collection variable.

This statement is always true – “put inverse=true in collection variable”, but do not blindfold on it, try to understand the reason behind is essential to optimal your Hibernate performance.

What is “inverse” ?

This is the most confusing keyword in Hibernate, at least i took quite a long time to understand it. The “inverse” keyword is always declare in **one-to-many** and **many-to-many** relationship (many-to-one doesn't has inverse keyword), it means which side is responsible to take care of the relationship.

“inverse”, should change to “relationship owner”?

In Hibernate, only the “relationship owner” should maintain the relationship, and the “inverse” keyword is created to defines which side is the owner to maintain the relationship. However the “inverse” keyword itself is not verbose enough, I would suggest change the keyword to “**relationship_owner**”.

In short, inverse=”true” means this is the relationship owner, and inverse=”false” (default) means it's not.

1. One to many Relationship

This is a **one-to-many** relationship table design, a STOCK table has many occurrences in STOCK_DAILY_RECORD table.

STOCK_ID
STOCK_CODE
STOCK_NAME

DAILY_RECORD_ID
PRICE_OPEN
PRICE_CLOSE
PRICE_CHANGE
VOLUME
DATE
STOCK_ID

2. Hibernate Implementation

See the Hibernate implementation in XML mapping files.

File : Stock.java

```
public class Stock implements java.io.Serializable {  
    ...  
    private Set<StockDailyRecord> stockDailyRecords =  
        new HashSet<StockDailyRecord>(0);  
    ...
```

File : StockDailyRecord.java

```
public class StockDailyRecord implements java.io.Serializable {  
    ...  
    private Stock stock;  
    ...
```

File : Stock.hbm.xml

```
<hibernate-mapping>  
    <class name="com.sekharit.entity.Stock" table="stock" ...>  
        ...  
        <set name="stockDailyRecords" table="stock_daily_record" fetch="select">  
            <key>  
                <column name="STOCK_ID" not-null="true" />  
            </key>  
            <one-to-many class="com.sekharit.entity.StockDailyRecord" />  
        </set>  
        ...
```

File : StockDailyRecord.hbm.xml

```
<hibernate-mapping>  
    <class name="com.sekharit.entity.StockDailyRecord" table="stock_daily_record" ...>  
        ...  
        <many-to-one name="stock" class="com.sekharit.entity.Stock">  
            <column name="STOCK_ID" not-null="true" />  
        </many-to-one>  
        ...
```

3. *inverse = true / false*

Inverse keyword is applied in one to many relationship. Here's the question, if save or update operation perform in "Stock" object, should it update the "stockDailyRecords" relationship?

File : Stock.hbm.xml

```
<class name="com.sekharit.entity.Stock" table="stock" ...>
...
<set name="stockDailyRecords" table="stock_daily_record" inverse="{true/false}" fetch="select">
  <key>
    <column name="STOCK_ID" not-null="true" />
  </key>
  <one-to-many class="com.sekharit.entity.StockDailyRecord" />
</set>
...
```

1. *inverse="true"*

If *inverse="true"* in the set variable, it means "stock_daily_record" is the relationship owner, so Stock will NOT UPDATE the relationship.

```
<class name="com.sekharit.entity.Stock" table="stock" ...>
...
<set name="stockDailyRecords" table="stock_daily_record" inverse="true" >
```

2. *inverse="false"*

If *inverse="false"* (default) in the set variable, it means "stock" is the relationship owner, and Stock will UPDATE the relationship.

```
<class name="com.sekharit.entity.Stock" table="stock" ...>
...
<set name="stockDailyRecords" table="stock_daily_record" inverse="false" >
```

See more examples below :

4. *inverse="false" Example*

If keyword "inverse" is not define, the *inverse = "false"* will be used, which is

```
<!--Stock.hbm.xml-->
<class name="com.sekharit.entity.Stock" table="stock" ...>
```

```
...<set name="stockDailyRecords" table="stock_daily_record" inverse="false">
```

It means "stock" is the relationship owner, and it will maintains the relationship.

Insert example ...

When a "Stock" object is saved, Hibernate will generated three SQL statements, two inserts and one update.

```
session.beginTransaction();

Stock stock = new Stock();
stock.setStockCode("7052");
stock.setStockName("PADINI");

StockDailyRecord stockDailyRecords = new StockDailyRecord();
stockDailyRecords.setPriceOpen(new Float("1.2"));
stockDailyRecords.setPriceClose(new Float("1.1"));
stockDailyRecords.setPriceChange(new Float("10.0"));
stockDailyRecords.setVolume(3000000L);
stockDailyRecords.setDate(new Date());

stockDailyRecords.setStock(stock);
stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);
session.save(stockDailyRecords);

session.getTransaction().commit();
```

Output...

Hibernate:

```
INSERT
INTO
    sekhardb.stock
(STOCK_CODE, STOCK_NAME)
VALUES
(?, ?)
```

Hibernate:

```
INSERT
INTO
    sekhardb.stock_daily_record
(STOCK_ID, PRICE_OPEN, PRICE_CLOSE, PRICE_CHANGE, VOLUME, DATE)
VALUES
(?, ?, ?, ?, ?, ?)
```

Hibernate:

```
UPDATE
    sekhardb.stock_daily_record
SET
    STOCK_ID=?
WHERE
    RECORD_ID=?
```

Stock will update the "stock_daily_record.STOCK_ID" through Set variable (stockDailyRecords), because Stock is the relationship owner.

Note

The third statement is really NOT necessary.

Update example ...

When a "Stock" object is updated, Hibernate will generate two SQL statements, one inserts and one update.

```
session.beginTransaction();
Stock stock = (Stock)session.get(Stock.class, 57);
StockDailyRecord stockDailyRecords = new StockDailyRecord();
stockDailyRecords.setPriceOpen(new Float("1.2"));
stockDailyRecords.setPriceClose(new Float("1.1"));
stockDailyRecords.setPriceChange(new Float("10.0"));
stockDailyRecords.setVolume(3000000L);
stockDailyRecords.setDate(new Date());
stockDailyRecords.setStock(stock);
stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stockDailyRecords);
session.update(stock);

session.getTransaction().commit();
```

Output...

Hibernate:

```
INSERT
INTO
    sekhardb.stock_daily_record
    (STOCK_ID, PRICE_OPEN, PRICE_CLOSE, PRICE_CHANGE, VOLUME, DATE)
VALUES
    (?, ?, ?, ?, ?, ?)
```

Hibernate:

```
UPDATE
    sekhardb.stock_daily_record
SET
    STOCK_ID=?
WHERE
    RECORD_ID=?
```

Note

Again, the third statement is NOT necessary.

5. *inverse="true"* Example

If keyword "inverse=true" is defined :

```
<!--Stock.hbm.xml-->
<class name="com.sekharit.entity.Stock" table="stock" ...>
    ...
        <set name="stockDailyRecords" table="stock_daily_record" inverse="true">
```

Now, it means "**stockDailyRecords**" is the relationship owner, and "stock" will not maintains the relationship.

Insert example ...

When a "Stock" object is saved, Hibernate will generated two SQL insert statements.

```
session.beginTransaction();

Stock stock = new Stock();
stock.setStockCode("7052");
stock.setStockName("PADINI");

StockDailyRecord stockDailyRecords = new StockDailyRecord();
stockDailyRecords.setPriceOpen(new Float("1.2"));
stockDailyRecords.setPriceClose(new Float("1.1"));
stockDailyRecords.setPriceChange(new Float("10.0"));
stockDailyRecords.setVolume(3000000L);
stockDailyRecords.setDate(new Date());

stockDailyRecords.setStock(stock);
stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);
session.save(stockDailyRecords);

session.getTransaction().commit();
```

Output ...

Hibernate:

```
INSERT  
INTO  
sekhardb.stock  
(STOCK_CODE, STOCK_NAME)  
VALUES  
(?, ?)
```

Hibernate:

```
INSERT  
INTO  
sekhardb.stock_daily_record  
(STOCK_ID, PRICE_OPEN, PRICE_CLOSE, PRICE_CHANGE, VOLUME, DATE)  
VALUES  
(?, ?, ?, ?, ?, ?)
```

Update example ...

When a "Stock" object is updated, Hibernate will generate one SQL statement.

```
session.beginTransaction();  
  
Stock stock = (Stock)session.get(Stock.class, 57);  
  
StockDailyRecord stockDailyRecords = new StockDailyRecord();  
stockDailyRecords.setPriceOpen(new Float("1.2"));  
stockDailyRecords.setPriceClose(new Float("1.1"));  
stockDailyRecords.setPriceChange(new Float("10.0"));  
stockDailyRecords.setVolume(3000000L);  
stockDailyRecords.setDate(new Date());  
  
stockDailyRecords.setStock(stock);  
stock.getStockDailyRecords().add(stockDailyRecords);  
  
session.save(stockDailyRecords);  
session.update(stock);  
  
session.getTransaction().commit();
```

Output...

Hibernate:

```
INSERT  
INTO  
sekhardb.stock_daily_record  
(STOCK_ID, PRICE_OPEN, PRICE_CLOSE, PRICE_CHANGE, VOLUME, DATE)
```

VALUES

(?, ?, ?, ?, ?, ?)

Conclusion

Understanding the "inverse" is essential to optimize your Hibernate code, it helps to avoid many unnecessary update statements, like "insert and update example for inverse=false" above. At last, try to remember the inverse="true" mean this is the relationship owner to handle the relationship.

Hibernate – Cascade example (save, update, delete and delete-orphan)

Cascade is a convenient feature to save the lines of code needed to manage the state of the other side manually.

The "Cascade" keyword is often appear on the collection mapping to manage the state of the collection automatically. In this tutorials, this one-to-many example will be used to demonstrate the cascade effect.

Cascade save / update example

In this example, if a 'Stock' is saved, all its referenced 'stockDailyRecords' should be saved into database as well.

1. No save-update cascade

In previous section, if you want to save the 'Stock' and its referenced 'StockDailyRecord' into database, you need to save both individually.

```
Stock stock = new Stock();
StockDailyRecord stockDailyRecords = new StockDailyRecord();
//set the stock and stockDailyRecords data

stockDailyRecords.setStock(stock);
stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);
session.save(stockDailyRecords);
```

Output

```
Hibernate:
    insert into neo.stock (STOCK_CODE, STOCK_NAME)
    values (?, ?)

Hibernate:
    insert into neo.stock_daily_record
    (STOCK_ID, PRICE_OPEN, PRICE_CLOSE, PRICE_CHANGE, VOLUME, DATE)
    values (?, ?, ?, ?, ?, ?)
```

2. With save-update cascade

The cascade="save-update" is declared in 'stockDailyRecords' to enable the save-update cascade effect.

```
<!-- Stock.hbm.xml -->
<set name="stockDailyRecords" cascade="save-update" table="stock_daily_record"...>
    <key>
        <column name="STOCK_ID" not-null="true" />
    </key>
    <one-to-many class="com.neo.common.StockDailyRecord" />
</set>
Stock stock = new Stock();
StockDailyRecord stockDailyRecords = new StockDailyRecord();
//set the stock and stockDailyRecords data

stockDailyRecords.setStock(stock);
stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);
Output
```

```
Hibernate:
insert into neo.stock (STOCK_CODE, STOCK_NAME)
values (?, ?)

Hibernate:
insert into neo.stock_daily_record
(STOCK_ID, PRICE_OPEN, PRICE_CLOSE, PRICE_CHANGE, VOLUME, DATE)
values (?, ?, ?, ?, ?, ?)
```

The code `session.save(stockDailyRecords);` is no longer required, when you save the 'Stock', it will "cascade" the save operation to its referenced 'stockDailyRecords' and save both into database automatically.

Cascade delete example

In this example, if a 'Stock' is deleted, all its referenced 'stockDailyRecords' should be deleted from database as well.

1. No delete cascade

You need to loop all the 'stockDailyRecords' and delete it one by one.

```
Query q = session.createQuery("from Stock where stockCode = :stockCode ");
q.setParameter("stockCode", "4715");
Stock stock = (Stock)q.list().get(0);

for (StockDailyRecord sdr : stock.getStockDailyRecords()){
    session.delete(sdr);
}
session.delete(stock);
Output
```

```
Hibernate:
delete from neo.stock_daily_record
where DAILY_RECORD_ID=?
```

```
Hibernate:
delete from neo.stock
where STOCK_ID=?
```

2. With delete cascade

The **cascade="delete"** is declared in 'stockDailyRecords' to enable the delete cascade effect. When you delete the 'Stock', all its reference 'stockDailyRecords' will be deleted automatically.

```
<!-- Stock.hbm.xml -->
<set name="stockDailyRecords" cascade="delete" table="stock_daily_record" ...>
    <key>
        <column name="STOCK_ID" not-null="true" />
    </key>
    <one-to-many class="com.neo.common.StockDailyRecord" />
</set>
```

```
Query q = session.createQuery("from Stock where stockCode = :stockCode ");
q.setParameter("stockCode", "4715");
Stock stock = (Stock)q.list().get(0);
session.delete(stock);
```

Output

```
Hibernate:
    delete from neo.stock_daily_record
    where DAILY_RECORD_ID=?
```

```
Hibernate:
    delete from neo.stock
    where STOCK_ID=?
```

Cascade delete-orphan example

In above cascade delete option, if you delete a Stock , all its referenced 'stockDailyRecords' will be deleted from database as well. How about if you just want to delete two referenced 'stockDailyRecords' records? This is called orphan delete, see example...

1. No delete-orphan cascade

You need to delete the 'stockDailyRecords' one by one.

```
StockDailyRecord sdr1 = (StockDailyRecord)session.get(StockDailyRecord.class,
                                                       new Integer(56));
StockDailyRecord sdr2 = (StockDailyRecord)session.get(StockDailyRecord.class,
                                                       new Integer(57));
```

```
session.delete(sdr1);
session.delete(sdr2);
```

Output

```
Hibernate:
    delete from neo.stock_daily_record
    where DAILY_RECORD_ID=?
```

```
Hibernate:
    delete from neo.stock_daily_record
    where DAILY_RECORD_ID=?
```

2. With delete-orphan cascade

The **cascade="delete-orphan"** is declared in 'stockDailyRecords' to enable the delete orphan cascade effect. When you save or update the Stock, it will remove those 'stockDailyRecords' which already mark as removed.

```
<!-- Stock.hbm.xml -->
<set name="stockDailyRecords" cascade="delete-orphan" table="stock_daily_record" >
    <key>
        <column name="STOCK_ID" not-null="true" />
    </key>
    <one-to-many class="com.neo.common.StockDailyRecord" />
</set>
StockDailyRecord sdr1 = (StockDailyRecord)session.get(StockDailyRecord.class,
                                                new Integer(56));
StockDailyRecord sdr2 = (StockDailyRecord)session.get(StockDailyRecord.class,
                                                new Integer(57));

Stock stock = (Stock)session.get(Stock.class, new Integer(2));
stock.getStockDailyRecords().remove(sdr1);
stock.getStockDailyRecords().remove(sdr2);

session.saveOrUpdate(stock);
```

Output

```
Hibernate:
    delete from neo.stock_daily_record
    where DAILY_RECORD_ID=?
Hibernate:
    delete from neo.stock_daily_record
    where DAILY_RECORD_ID=?
```

In short, delete-orphan allow parent table to delete few records (delete orphan) in its child table.

How to enable cascade ?

The cascade is supported in both XML mapping file and annotation.

1. XML mapping file

In XML mapping file, declared the cascade keyword in your relationship variable.

```
<!-- Stock.hbm.xml -->
<set name="stockDailyRecords" cascade="save-update, delete"
      table="stock_daily_record" ...>
    <key>
        <column name="STOCK_ID" not-null="true" />
    </key>
    <one-to-many class="com.neo.common.StockDailyRecord" />
</set>
```

2. Annotation

In annotation, declared the CascadeType.SAVE_UPDATE (save, update) and CascadeType.REMOVE (delete) in @Cascade annotation.

```
//Stock.java
@OneToMany(mappedBy = "stock")
@Cascade({CascadeType.SAVE_UPDATE, CascadeType.DELETE})
public Set<StockDailyRecord> getStockDailyRecords() {
    return this.stockDailyRecords;
}
```

Further study – Cascade – JPA & Hibernate annotation common mistake.

inverse vs cascade

Both are totally different notions, see the differential here.

Conclusion

Cascade is a very convenient feature to manage the state of the other side automatically. However this feature come with a price, if you do not use it wisely (update or delete), it will generate many unnecessary cascade effects (cascade update) to slow down your performance, or delete (cascade delete) some data you didn't expected.

- 1) **cascade="none"**, the default, tells Hibernate to ignore the association.
- 2) **cascade="save-update"** tells Hibernate to navigate the association when the transaction is committed and when an object is passed to save() or update() and save newly instantiated transient instances and persist changes to detached instances.
- 3) **cascade="delete"** tells Hibernate to navigate the association and delete persistent instances when an object is passed to delete().
- 4) **cascade="all"** means to cascade both save-update and delete, as well as calls to evict and lock.
- 5) **cascade="all-delete-orphan"** means the same as cascade="all" but, in addition, Hibernate deletes any persistent entity instance that has been removed (dereferenced) from the association (for example, from a collection).
- 6) **cascade="delete-orphan"** Hibernate will delete any persistent entity instance that has been removed (dereferenced) from the association (for example, from a collection).

Different between cascade and inverse

Many Hibernate developers are confuse about the cascade option and inverse keyword. In some ways..they really look quite similar at the beginning, both are related with relationship.

Cascade vs inverse

However, there is no relationship between cascade and inverse, both are totally different notions.

1. inverse

This is used to decide which side is the relationship owner to manage the relationship (insert or update of the foreign key column).

Example

In this example, the relationship owner is belong to stockDailyRecords (inverse=true).

```
<!-- STOCK.DDM.XML -->
<hibernate-mapping>
```

```
<class name="com.neo.common.Stock" table="stock" ...>
...
<set name="stockDailyRecords" table="stock_daily_record" inverse="true">
    <key>
        <column name="STOCK_ID" not-null="true" />
    </key>
    <one-to-many class="com.neo.common.StockDailyRecord" />
</set>
...
```

When you save or update the stock object

```
session.save(stock);
session.update(stock);
```

Hibernate will only insert or update the STOCK table, no update on the foreign key column. [More detail example here...](#)

2. cascade

In cascade, after one operation (save, update and delete) is done, it decide whether it need to call other operations (save, update and delete) on another entities which has relationship with each other.

Example

In this example, the cascade="save-update" is declare on stockDailyRecords.

```
<!-- Stock.hbm.xml -->
<hibernate-mapping>
    <class name="com.neo.common.Stock" table="stock" ...>
    ...
        <set name="stockDailyRecords" table="stock_daily_record"
            cascade="save-update" inverse="true">
            <key>
                <column name="STOCK_ID" not-null="true" />
            </key>
            <one-to-many class="com.neo.common.StockDailyRecord" />
        </set>
    ...

```

When you save or update the stock object

```
session.save(stock);
session.update(stock);
```

It will inserted or updated the record into STOCK table and call another insert or update statement (cascade="save-update") on StockDailyRecord. [More detail example here...](#)

Conclusion

In short, the "inverse" is decide which side will update the foreign key, while "cascade" is decide what's the follow by operation should execute. Both are look quite similar in relationship, but it's totally two different things. Hibernate developers are worth to spend time to research on it, because misunderstand the concept or misuse it will bring serious performance or data integrity issue in your application.

Hibernate Dynamic SQL Generation

By default, Hibernate creates SQL statements for each persistent class on startup. These statements are simple create, read, update, and delete operations for reading a single row, deleting a row, and so on.

In some situations, such as a legacy table with hundreds of columns where the SQL statements will be large for most of the columns, and (say, only one column needs updating), you have to turn off this startup SQL generation and switch to dynamic statements generated at runtime. An extremely large number of entities can also impact startup time, because Hibernate has to generate all SQL statements for CRUD upfront. Memory consumption for this query statement cache will also be high if a dozen statements must be cached for thousands of entities (this isn't an issue, usually).

The default value of **dynamic-insert** and **dynamic-update** is **false**, which means include null properties in the Hibernate's SQL INSERT and SQL UPDATE statement. The **dynamic-insert** attribute tells Hibernate whether to include null property values in an SQL INSERT, and the **dynamic-update** attribute tells Hibernate whether to include unmodified properties in the SQL UPDATE.

You can configure the **dynamic-insert** and **dynamic-update** properties value through annotation or XML mapping file.

Hibernate – dynamic-insert attribute example

What is dynamic-insert

The dynamic-insert attribute tells Hibernate whether to include null properties in the SQL INSERT statement. Let explore some examples to understand more clear about it.

Dynamic-insert example

1. **dynamic-insert=false**

The default value of dynamic-insert is false, which means include null properties in the Hibernate's SQL INSERT statement.

For example, try set some null values to an object properties and save it.

```
StockTransaction stockTran = new StockTransaction();
//stockTran.setPriceOpen(new Float("1.2"));
//stockTran.setPriceClose(new Float("1.1"));
//stockTran.setPriceChange(new Float("10.0"));
```

```

stockTran.setVolume(2000000L);
stockTran.setDate(new Date());
stockTran.setStock(stock);

```

```
session.save(stockTran);
```

Turn on the Hibernate "show_sql" to true, you will see the following insert SQL statement.

```

Hibernate:
  INSERT
  INTO
    neo.stock_transaction
    (DATE, PRICE_CHANGE, PRICE_CLOSE, PRICE_OPEN, STOCK_ID, VOLUME)
  VALUES
    (?, ?, ?, ?, ?, ?)

```

Hibernate will generate the unnecessary columns (PRICE_CHANGE, PRICE_CLOSE, PRICE_OPEN) for the insertion.

2. dynamic-insert=true

If set the dynamic-insert to true, which means exclude null property values in the Hibernate's SQL INSERT statement.

For example, try set some null values to an object properties and save it again.

```

StockTransaction stockTran = new StockTransaction();
//stockTran.setPriceOpen(new Float("1.2"));
//stockTran.setPriceClose(new Float("1.1"));
//stockTran.setPriceChange(new Float("10.0"));
stockTran.setVolume(2000000L);
stockTran.setDate(new Date());
stockTran.setStock(stock);

session.save(stockTran);

```

Turn on the Hibernate "show_sql" to true. You will see the different insert SQL statement.

```

Hibernate:
  INSERT
  INTO
    neo.stock_transaction
    (DATE, STOCK_ID, VOLUME)
  VALUES
    (?, ?, ?)

```

Hibernate will generate only the necessary columns (DATE, STOCK_ID, VOLUME) for the insertion.

Performance issue

In certain situations, such as a very large table with hundreds of columns (legacy design), or a table contains extremely large data volume, insert something not necessary definitely will drop down your system performance.

How to configure it

You can configure the dynamic-insert properties value through annotation or XML mapping file.

1. Annotation

```
@Entity  
@Table(name = "stock_transaction", catalog = "neo")  
@org.hibernate.annotations.Entity(  
    dynamicInsert = true  
)  
public class StockTransaction implements java.io.Serializable {
```

2. XML mapping

```
<class ... table="stock_transaction" catalog="neo" dynamic-insert="true">  
    <id name="tranId" type="java.lang.Integer">  
        <column name="TRAN_ID" />  
        <generator class="identity" />  
    </id>
```

Conclusion

This little “dynamic-insert” tweak may increase your system performance, and highly recommends to do it. However one question in my mind is why Hibernate set it to false by default?

Hibernate – dynamic-update attribute example

What is dynamic-update

The dynamic-update attribute tells Hibernate whether to include unmodified properties in the SQL UPDATE statement.

Dynamic-update example

1. dynamic-update=false

The default value of dynamic-update is false, which means **include unmodified properties** in the Hibernate's SQL update statement.

For example, get an object and try modify its value and update it.

```
Query q = session.createQuery("from StockTransaction where tranId = :tranId ");  
q.setParameter("tranId", 11);  
StockTransaction stockTran = (StockTransaction)q.list().get(0);  
  
stockTran.setVolume(4000000L);  
session.update(stockTran);
```

Hibernate will generate the following update SQL statement.

```
Hibernate:  
    UPDATE  
        neo.stock_transaction  
    SET
```

```
DATE=?,
PRICE_CHANGE=?,
PRICE_CLOSE=?,
PRICE_OPEN=?,
STOCK_ID=?,
VOLUME=?

WHERE
TRAN_ID=?
```

Hibernate will update all the unmodified columns.

2. dynamic-update=true

If set the dynamic-insert to true, which means exclude unmodified properties in the Hibernate's SQL update statement.

For example, get an object and try modify its value and update it again.

```
Query q = session.createQuery("from StockTransaction where tranId = :tranId ");
q.setParameter("tranId", 11);
StockTransaction stockTran = (StockTransaction)q.list().get(0);

stockTran.setVolume(4000000L);
session.update(stockTran);
```

Hibernate will generate different update SQL statement.

```
hibernate:
UPDATE
    neo.stock_transaction
SET
    VOLUME=?
WHERE
    TRAN_ID=?
```

Hibernate will update the modified columns only.

Performance issue

In a large table with many columns (legacy design) or contains large data volumes, update some unmodified columns are absolutely unnecessary and great impact on the system performance.

How to configure it

You can configure "dynamic-update" properties via annotation or XML mapping file.

1. Annotation

```
@Entity
@Table(name = "stock_transaction", catalog = "neo")
@org.hibernate.annotations.Entity(
    dynamicUpdate = true
)
public class StockTransaction implements java.io.Serializable {
```

2. XML mapping

```
<class ... table="stock_transaction" catalog="neo" dynamic-update="true">
    <id name="tranId" type="java.lang.Integer">
```

```

<column name="TRAN_ID" />
<generator class="identity" />
</id>

```

Conclusion

This little “dynamic-update” tweak will definitely increase your system performance, and highly recommended to do it.

How to use database reserved keyword in Hibernate ?

In Hibernate, when you try to save an object into a table with any database reserved keyword as column name, you may hit the following error ...

```

ERROR JDBCExceptionReporter:78 - You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server version for the
right syntax to use near 'Datadabase reserved keyword....'

```

Reserved keyword “DESC”

In MySQL, “DESC” is the reserved keyword. Let see some examples to demonstrate how to use this reserved keyword in Hibernate.

Hibernate XML Mapping file

This is the default XML mapping file implementation for a table column, it will cause JDBCException...

```

<property name="desc" type="string" >
    <column name="DESC" length="255" not-null="true" />
</property>

```

Solution

1. Enclose the keyword with square brackets [].

```

<property name="desc" type="string" >
    <column name="[DESC]" length="255" not-null="true" />
</property>

```

2. Use single quote(') to enclose the double quotes ("")

```

<property name="desc" type="string" >
    <column name='"DESC"' length="255" not-null="true" />
</property>

```

Hibernate Annotation

This is the default annotation implementation for a table column, it will cause JDBCException...

```

@Column(name = "DESC", nullable = false)
public String getdesc() {
    return this.desc;
}

```

Solution

1. Enclose the keyword with square brackets [].

```
@Column(name = "[DESC]", nullable = false)
public String getDesc() {
    return this.desc;
}
```

2. Use double quotes ("") to enclose it.

```
@Column(name = "\"DESC\"", nullable = false)
public String getDesc() {
    return this.desc;
}
```

Conclusion

This same solution can also apply to the reserved keyword as table name.