

# Re-Write **Hibernate**

<b>1. INTRODUCTION .....</b>	<b>4</b>
<b>JDBC vs HIBERNATE.....</b>	<b>4</b>
HIBERNATE IMPLEMENTED BASED ON PROXY TEMPLATE.....	4
HIBERNATE ARCHITECTURE.....	5
1. Configuration .....	6
2. SessionFactory Object .....	8
3. Session Object.....	8
4. Transaction Object .....	9
5. Query Object .....	9
6. Criteria Object .....	9
<b>2. HIBERNATE EXAMPLE STEP BY STEP .....</b>	<b>9</b>
1. CHOOSE DATABASE TABLE.....	9
2. POJO CLASS / PERSISTENCE CLASS .....	10
3. MAPPING FILE.....	10
4. CONFIGURATION FILE .....	11
5. APPLICATION CLASS FOR MAIN LOGIC.....	12
1. Load Hibernate API into Our Application.....	12
2. Load Configurations.....	12
3. Open Session .....	12
4. Create Transaction Object .....	13
5. Close the Connections .....	13
<b>3. HIBERNATE CURD OPERATIONS.....</b>	<b>14</b>
1. Choose Database Table.....	15
2. POJO class( <i>EmployeeBo.java</i> ) .....	15
3. O/R Mapping XML( <i>EmployeeBo.hbm.xml</i> ) .....	16
4. Hibernate Configuration file.....	16
1. SELECT OPERATION USING HIBERNATE .....	16
2. INSERT OPERATION USING HIBERNATE .....	18
3. UPDATE OPERATION USING HIBERNATE.....	21
Approach 1:.....	21
Approach 2:.....	21
What is the difference between and merge and update?.....	22
4. DELETE OPERATION USING HIBERNATE.....	24
5. CLEAR .....	24
<b>4. HIBERNATE POJO CLASS LIFECYCLE .....</b>	<b>26</b>
1. Transient state.....	26
2. Persistent state.....	26

3. Detached state .....	26
<b>5. HIBERNATE INHERITANCE MAPPING.....</b>	<b>27</b>
1. TABLE PER CLASS HIERARCHY.....	30
2. TABLE PER SUB-CLASS HIERARCHY.....	31
3. TABLE PER CONCRETE CLASS HIERARCHY.....	33
<b>6. HIBERNATE GENERATORS &lt;GENERATOR&gt;.....</b>	<b>34</b>
LIST OF GENERATORS .....	34
1. <i>assigned</i> .....	35
2. <i>increment</i> .....	35
3. <i>identity</i> .....	35
4. <i>sequence</i> .....	36
5. <i>bilo</i> .....	36
6. <i>native</i> .....	37
7. <i>foreign</i> .....	37
8. <i>uuid</i> .....	37
Custom generator .....	37
GENERATORS EXAMPLE .....	38
<b>7. HIBERNATE QUERY LANGUAGE (HQL) .....</b>	<b>40</b>
SQL VS HQL .....	40
QUERY INTERFACE .....	41
HQL EXAMPLES .....	41
1. <i>HQL Select Query Example</i> .....	43
2. <i>HQL Update/Delete Query Example</i> .....	44
3. <i>HQL Insert Query Example</i> .....	45
HQL WITH AGGREGATE FUNCTIONS.....	45
<b>8. HIBERNATE CRITERIA QUERY LANGUAGE(HCQL) .....</b>	<b>46</b>
Criteria Interface.....	46
WORKING WITH CRITERIA QUERY LANGUAGE .....	46
Example: Get the details from bank table where balance>3000. ....	47
Example: Adding ORDERBY Conditions to Criteria .....	49
Example: Adding multiple conditions.....	49
HIBERNATE PROJECTIONS.....	50
Projections Syntax .....	50
Example 1: Load Single Column using Projections .....	50
Example 2: Load Multiple Columns using Projections.....	51
Difference between HQL and CQL .....	52
<b>9. NATIVE SQL QUERIES .....</b>	<b>52</b>
WORKING WITH NATIVE SQL QUERIES.....	52
Example .....	53
<b>10. NAMED QUERIES .....</b>	<b>54</b>
Example 1 : <i>HQL Named Query Example</i> .....	54
Example 2 : <i>Native SQL Named Query Example</i> .....	56
<b>11. HIBERNATE RELATIONSHIPS .....</b>	<b>57</b>

RELATIONSHIPS IN HIBERNATE.....	58
<i>Mapping Directions.....</i>	58
1.ONE-TO-ONE RELATIONSHIP MAPPING EXAMPLE.....	59
2.ONE-TO-MANY / MANY-TO-ONE RELATIONSHIP.....	62
3.MANY TO MANY RELATIONSHIP.....	65
<i>Cascade Attribute In Hibernate .....</i>	67
<b>12. HIBERNATE CACHE .....</b>	<b>68</b>
1. <i>First Level Cache .....</i>	68
2. <i>Second Level Cache .....</i>	69
<b>13. HIBERNATE WITH ANNOTATIONS.....</b>	<b>72</b>
COMMONLY USED ANNOTATIONS IN HIBERNATE .....	72
EXAMPLE: CURD OPERATIONS USING ANNOTATIONS.....	73
MAPPINGS USING ANNOTATIONS .....	76
1. <i>one-to-one mapping using Annotations.....</i>	76
2. <i>one-to-many Mapping Using Annotations .....</i>	78
3. <i>Many-to-many Mapping Using Annotations .....</i>	80
<b>ERRORS &amp; SOLUTIONS .....</b>	<b>82</b>
ORG.HIBERNATE.HIBERNATEEXCEPTION: COULD NOT PARSE CONFIGURATION: HIBERNATE.CFG.XML.....	82
<b>REFERENCES .....</b>	<b>82</b>
<b>HIBERNATE INTERVIEW QUESTIONS .....</b>	<b>83</b>
1.2 <i>What are advantages of Hibernate?.....</i>	83
2 <i>What is caching? .....</i>	83
3 <i>What are some core interfaces of hibernate?.....</i>	83
4 <i>Difference between get() vs load() method in Hibernate? (detailed answer) The key difference between get() and load() method is that</i> 83	
5 <i>What is the difference between save() and persist() method in Hibernate? .....</i>	84
6 <i>What is the difference between and merge and update? .....</i>	84
7 <i>Different between cascade and inverse .....</i>	84
8 <i>Does SessionFactory is thread-safe in Hibernate? (.....</i>	86
9 <i>Does Hibernate Session interface is thread-safe in Java? (detailed answer) .....</i>	86
<i>What is difference between getCurrentSession() and openSession() in Hibernate?.....</i>	86
<i>openSession().....</i>	86
<i>getCurrentSession().....</i>	86
<i>Can you declare Entity(Bean) class as final in hibernate?.....</i>	86
<i>Does entity class (Bean) in hibernate require no arg constructor?.....</i>	86
<i>How do you log SQL queries issued by the Hibernate framework in Java application? .....</i>	87
<i>What is named SQL query in Hibernate? .....</i>	87
<i>Explain Criteria API.....</i>	87
<i>How do you switch between relational databases without code changes? .....</i>	88
<i>What is Hibernate proxy?.....</i>	88
<i>What is automatic dirty checking? .....</i>	88
<i>What is query cache in Hibernate? .....</i>	89
<i>What are two types of Collections in hibernate?.....</i>	89
<i>What is lazy loading in hibernate?.....</i>	89

# 1. Introduction

Hibernate is the ORM tool given to transfer the data between a java (object) application and a database (Relational) in the form of the objects. Hibernate is the open source light weight tool given by **Gavin King**.

Hibernate is a non-invasive framework, means it won't force the programmers to extend/implement any class/interface, and in hibernate we have all POJO classes so its light weight

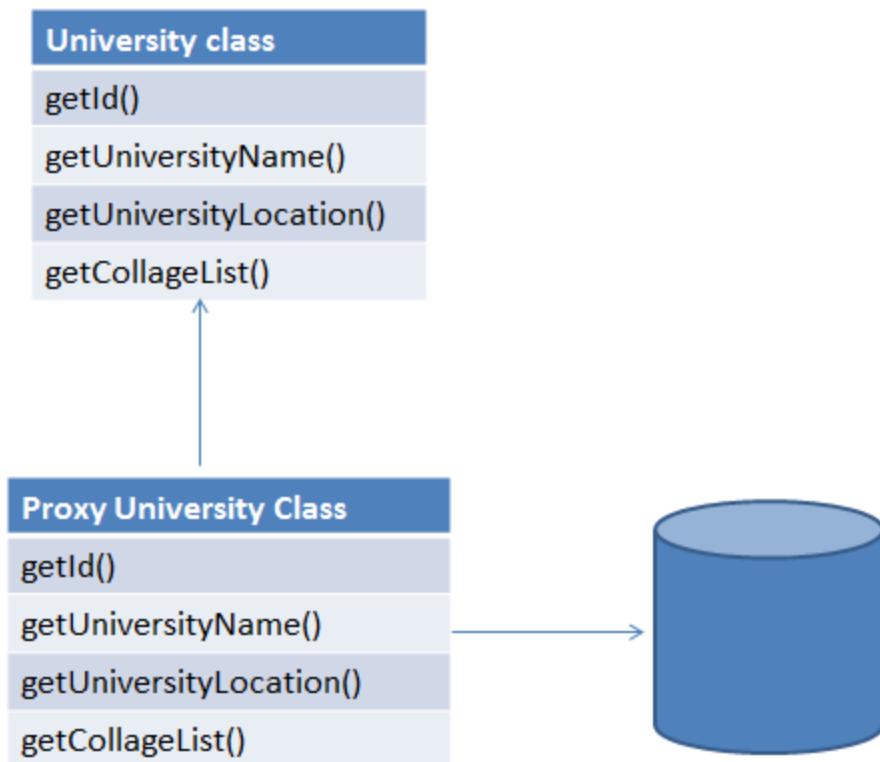
## JDBC vs HIBERNATE

JDBC	HIBERNATE
Programmer must close the connection. Jdbc doesn't responsible to close the connection.	Hibernate will take care about closing the connections
if the table structure is modified then the JDBC program doesn't work, again we need to modify and compile the programs	Hibernate has its own query language(HQL) and it is database independent. So if we change the database, then also our application will work, because HQL is database independent
In jdbc all exceptions are checked exceptions, so we must write code in try, catch and throws	hibernate we have only Un-checked exceptions, so no need to write try, catch, or throws
JDBC won't generate primary keys automatically	Hibernate has capability to generate primary keys automatically while we are storing the records into database.
JDBC won't support Caching mechanism	Hibernate supports caching mechanism by this, the number of round trips between an application and the database will be reduced, by using this caching technique an application performance will be increased automatically.
In JDBC , we need to write SQL queries manually	Hibernate provided Dialect classes, so we no need to write sql queries in hibernate, instead we use the methods provided by that API

## Hibernate implemented based on Proxy Template

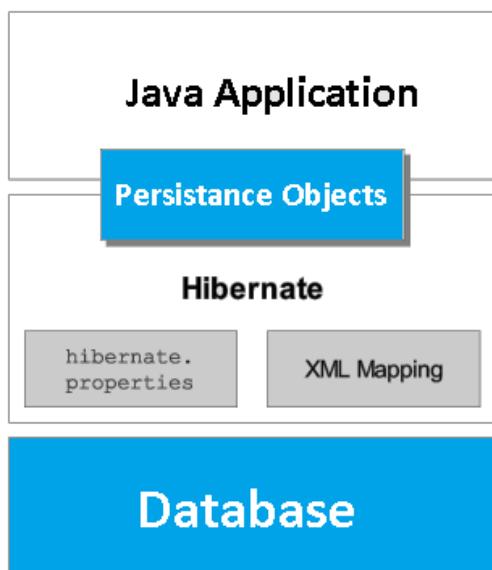
- Hibernate doesn't hit the database always, it returns Proxy object if data is already available in cache.
- If data not available, proxy will hit the database, and stores the data in Proxy object & return Proxy object to application/client.

- The proxy attribute enables lazy initialization of persistent instances of the class. Hibernate will initially return CGLIB proxies which implement the named interface. The actual persistent object will be loaded when a method of the proxy is invoked.

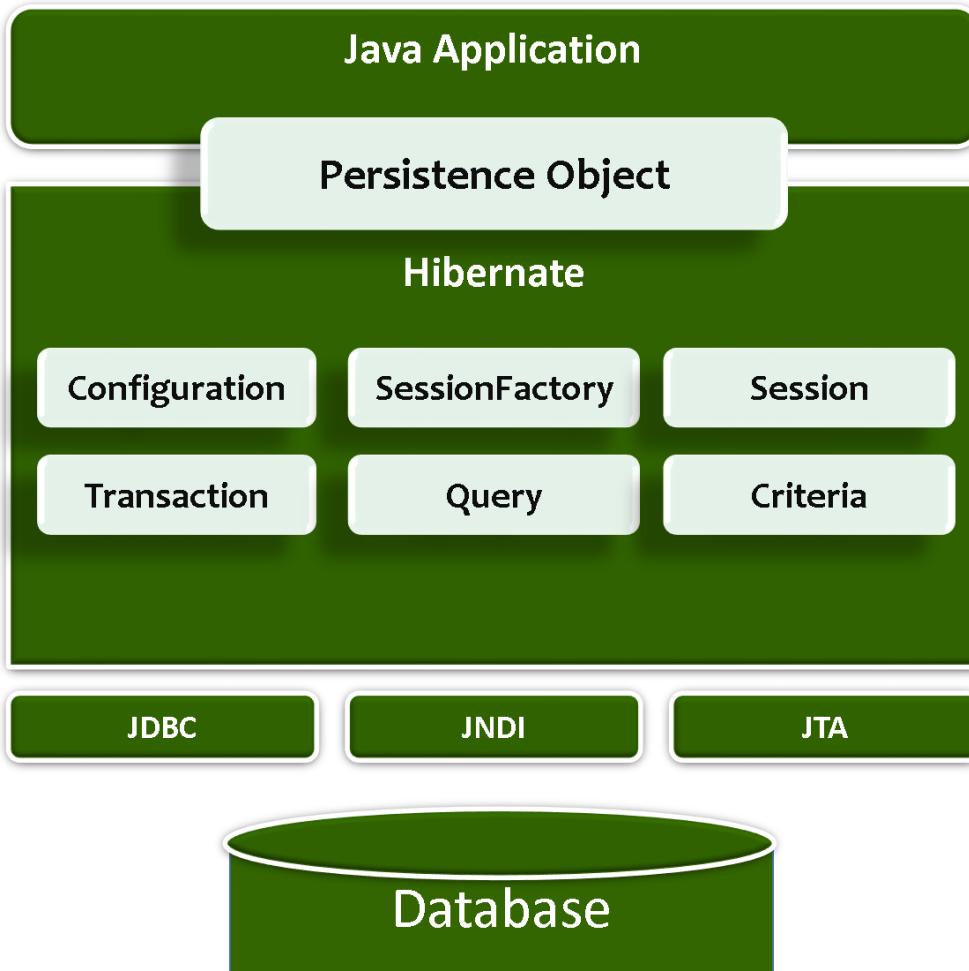


## Hibernate Architecture

The diagram below provides a high-level view of the Hibernate architecture:



Following is a detailed view of the Hibernate Application Architecture



Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JTA), and Java Naming and Directory Interface (JNDI) along with its own API objects like SessionFactory, Session, Transaction etc.,

## 1. Configuration

It represents a **configuration or properties file required by the Hibernate**. Configuration is the file loaded into hibernate application when working with hibernate.

Configuration file contains 3 types of information.

- i. **Connection Properties**
- ii. **Hibernate Properties**
- iii. **Mapping file name(s)**

```

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

<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/mydb</property>
        <property name="connection.username">root</property>
        <property name="connection.password">root</property>
        <property name="connection.pool_size">1</property>

        <!-- Hibernate Properties -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">validate</property>

        <!-- Mapping file name(s)-->
        <mapping resource="res/employee.hbm.xml"/>
    </session-factory>
</hibernate-configuration>

```

### How do you switch between relational databases without code changes?

Using Hibernate SQL Dialects, we can switch databases. Hibernate will generate appropriate hql queries based on the dialect defined.

### Hibernate dialect class vs Driver

**Driver:** A database driver is a program for which implements a protocol (ODBC, JDBC) for connecting to a database. It is an Adaptor which connects a generic interface to a specific vendor's implementation, just like printer drivers etc.

**"Driver is like English"**

**Dialect :** A database dialect is a configuration setting for platform independent software (JPA, Hibernate, etc) which allows such software to translate its generic SQL statements into vendor specific DDL, DML.

**"Dialect is the different pronunciations of English"**

We all know there are different versions of Oracle... Oracle 9i, Oracle8, Oracle 10g. The driver we would use would be a common for all of these. But the dialect we use is specific to each one of them, which helps Hibernate in generating optimized queries to those specific versions of database and also this is not mandatory to be given in cfg.xml.

This is the SQL dialect (database type) for the database being used.

For connecting any hibernate application with the database, you must specify the SQL dialects. There are many Dialects classes defined for RDBMS in the org.hibernate.dialect package. They are as follows:

RDBMS	Dialect
Oracle (any version)	<code>org.hibernate.dialect.OracleDialect</code>
Oracle9i	<code>org.hibernate.dialect.Oracle9iDialect</code>
Oracle10g	<code>org.hibernate.dialect.Oracle10gDialect</code>
MySQL	<code>org.hibernate.dialect.MySQLDialect</code>
MySQL with InnoDB	<code>org.hibernate.dialect.MySQLInnoDBDialect</code>
MySQL with MyISAM	<code>org.hibernate.dialect.MySQLMyISAMDialect</code>
DB2	<code>org.hibernate.dialect.DB2Dialect</code>
DB2 AS/400	<code>org.hibernate.dialect.DB2AS400Dialect</code>
DB2 OS390	<code>org.hibernate.dialect.DB2OS390Dialect</code>
Microsoft SQL Server	<code>org.hibernate.dialect.SQLServerDialect</code>
SAP DB	<code>org.hibernate.dialect.SAPDBDialect</code>
Informix	<code>org.hibernate.dialect.InformixDialect</code>

## 2. SessionFactory Object

The `org.hibernate.SessionFactory` interface provides factory method **to get the object of Session**

It holds second level cache (optional) of data like **Dialect, username and password**. All these data are fixed for whole application, so it is **immutable**. And if many threads access at same time there is no change in data, so it is **thread safe**.

## 3. Session Object

It Opens a Session between Database and our Application. It holds a **first-level cache (mandatory)** of data. The `org.hibernate.Session` interface provides **methods to insert, update and delete** the object

Session is a light weight and a non-threadsafe object (No, you cannot share it between threads) that represents a single unit-of-work with the database

## 4. Transaction Object

The **org.hibernate.Transaction** interface provides methods for transaction management.

## 5. Query Object

Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects.

## 6. Criteria Object

Criteria object are used to create and execute object oriented criteria queries to retrieve objects.

# 2. Hibernate Example Step by Step

For Developing any hibernate application we need create below 4 files always

1. Choose Database Table
2. POJO class
3. Mapping XML
4. Configuration XML
5. Application class for main logic

Above are the minimum requirement to run any hibernate application, but we may create any number of POJO classes and any number of mapping xml files (**Number of POJO classes = that many number of mapping xmls**), and only **one configuration xml** and finally **one java file to write our logic**.

### 1. Choose Database Table

In this Example we are taking '**employee**' table of 'mydb' database.

```
CREATE TABLE `employee` (
`eid` INT(11) NOT NULL AUTO_INCREMENT,
`name` VARCHAR(50) NOT NULL DEFAULT '0',
`address` VARCHAR(50) NOT NULL DEFAULT '0',
PRIMARY KEY (`eid`)
)
COLLATE='latin1_swedish_ci' ENGINE=InnoDB AUTO_INCREMENT=5;
```

```
mysql> select * from employee;
+----+-----+-----+
| eid | name | address |
+----+-----+-----+
| 1  | SATYA | VIJAYAWADA |
| 2  | SURYA | HYDERABAD |
| 3  | RAVI  | PUNE   |
+----+-----+-----+
```

## 2. POJO class / Persistence class

- **POJO is a simple java file**, no need to extend any class or implement any interface.
- Pojo class contains **table column names as data members**,
- In above table eid, name, address are columns
- **It contains column names as private data members with setters and getters**

### **EmployeeBo.java**

```
public class EmployeeBo {  
    private int eid;  
    private String name;  
    private String address;  
    //setters / getters  
}
```

## 3. Mapping File

- Mapping file contains **mapping from a pojo class name to a table name and pojo class variable names to table column names**.
- We can create any no. of mapping files
- Mapping can be done using 2 ways, using XML & using Annotations.

### Syntax of mapping xml (Pojaclassname.hbm.xml)

```
<hibernate-mapping>  
<class name="POJO class name" table="table name in database">  
    <id name="variable name" column="column name in db" type="java/hib type" />  
    <property name="var1 name" column="column name in db" type="java/hib type" />  
    <property name="var2 name" column="column name in db" type="java/hib type" />  
</class>  
</hibernate-mapping>
```

### Example: EmployeeBo.hbm.xml

```
<hibernate-mapping>  
    <class name="bo.EmployeeBo" table="employee">  
        <id name="eid" column="eid">  
            <generator class="assigned" />  
        </id>  
        <property name="name" column="name" />  
        <property name="address" column="address" />  
    </class>  
</hibernate-mapping>
```

### Explanation

- **<hibernate-mapping>** is the root element in the mapping file.
- **<class>** specifies the Persistent/POJO class.
- **<id>** specifies the primary key attribute in the class.
- **<generator>** is the sub element of id. It is used to generate the primary key. There are many generator classes such as assigned (It is used if id is specified by the user), increment, hilo, sequence, native etc. We will learn all the generator classes later.
- **property** It is the subelement of class that specifies the property name of the Persistent class.

## 4. Configuration File

- Configuration file contains 3 types of information.
  - i. **Connection Properties**
  - ii. **Hibernate Properties**
  - iii. **Mapping file name(s)**
- **We must create configuration file for each database we are going to use.**

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

### Syntax of Configuration file

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

    <!-- Related to the connection START -->
    <property name="connection.driver_class">Driver Class Name </property>
    <property name="connection.url">URL </property>
    <property name="connection.user">user </property>
    <property name="connection.password">password</property>
    <!-- Related to the connection END -->

    <!-- Related to hibernate properties START -->
    <property name="show_sql">true/false</property>
    <property name="dialet">Database dialect class</property>
    <property name="hbm2ddl.auto">create/update or what ever</property>
    <!-- Related to hibernate properties END-->

    <!-- Related to mapping START-->
    <mapping resource="hbm file 1 name .xml" />
    <mapping resource="hbm file 2 name .xml" />
    <!-- Related to the mapping END -->

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

### Example: hibernate.cfg.xml

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

<hibernate-configuration>
<session-factory>
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/smlcodes</property>
<property name="hibernate.connection.username">root</property>
<property name="hibernate.connection.password">root</property>

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

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

## 5. Application class for main logic

We will create our application class with the **main()** method to run the application. We will use this application to save Employee data. Please follow the steps to write Application class

### **1. Load Hibernate API into Our Application**

Load hibernate API by writing these two lines on the top of the application.

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

### **2. Load Configurations**

Among **Configuration(hibernate.cfg.xml)**, **Mapping xml files**, first we need to load configuration xml, because once we load the configuration file, automatically mapping xml file will also load, because it is defined it **Configuration(hibernate.cfg.xml) file**

We can load the configuration file by using **Configuration** class Object. **cfg.configure("xml")** loads the all the configurations from config file and save in **config object, now config object contains all configuration details.**

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

### **3. Open Session**

SessionFactory will produces the Session objects based on the requests. To get Session Object we have to call **openSession()** method on Sessionfactory Object.

Usually an application has a single SessionFactory instance and threads servicing client requests obtain Session instances from this factory.

**Whenever session is opened then internally a database connection will be opened**

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

## 4. Create Transaction Object

While working with **insert, update, delete, operations** from a hibernate application onto the database then hibernate needs a logical Transaction, if we are **selecting an object** from the database then we do **not require any logical transaction** in hibernate.

To begin a logical transaction in hibernate then we need to call a method **beginTransaction()** given by Session Interface

```
Transaction tx = session.beginTransaction();
session.save(bo);
System.out.println("Employee Data saved successfully.....!!!");
tx.commit();
```

We have following methods for performing CURD operations

<b>session.save(bo)</b>	-	Inserting object "bo" into database
<b>session.update(bo)</b>	-	Updating object "bo" in the database
<b>session.load(bo)</b>	-	Selecting object "bo" object
<b>session.delete(bo)</b>	-	Deleting object "bo" from database

Finally we need to call **commit()** in Transaction, like **tx.commit()**;

## 5. Close the Connections

Finally we need to close the all open connections

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

### Example : EmployeeSave.java

```
package app;

import org.hibernate.*;
import org.hibernate.cfg.*;
import bo.EmployeeBo;

public class EmployeeSave {
    public static void main(String[] args) {
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

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

        EmployeeBo bo = new EmployeeBo();
        bo.setEid(5);
        bo.setName("DILEEP");
        bo.setAddress("BANGLORE");
```

```

        Transaction tx = session.beginTransaction();
        session.save(bo);
        System.out.println("Employee Data saved successfully.....!!!");
        tx.commit();
        session.close();
        factory.close();
    }
}

```

```

log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.Environment).
log4j:WARN Please initialize the log4j system properly.
Employee Data saved successfully.....!!
Hibernate: insert into employee (name, address, eid) values (?, ?, ?)

```

### Summarized Steps for creating Application class

**Configuration**

**SessionFactory**

**Session**

**Transaction**

**Close Statements**

## 3. Hibernate CURD Operations

Now we will see how to insert, delete, update & select data from/into database using hibernate.  
Session class will have the following methods to perform CURD Operations

### Session class methods

#### Select

```

Object get(Class , Serializable id)
Object load(Class, Serializable id)

```

#### Insert

```

Serializable save(Object object)
void      persist(Object object)
void      saveOrUpdate(Object object)

```

#### Update

```

Object merge(Object object)
void      update(Object object)

```

### Delete

```
void delete(Object object)
```

### Clear

```
void evict(Object object) : Remove this instance from the session cache.  
void clear() : Completely clear the session.
```

### Other

```
boolean isDirty() : Does this session contain any changes which must be synchronized with the database? In other words, would any DML operations be executed if we flushed this session?
```

```
void refresh(Object object)
```

Re-read the state of the given instance from the underlying database.

Below files are common to all CURD operation examples, follow the Steps

## 1. Choose Database Table

```
mysql> select * from employee;  
+----+-----+-----+  
| eid | name | address |  
+----+-----+-----+  
| 1 | Satya | VIJYAYAWADA |  
| 2 | Ravi | HYDERABAD |  
| 3 | SURYA | HYDERABAD |  
| 4 | RAMAN | PUNE |  
| 5 | DILEEP | BANGLORE |  
| 6 | DILEEP | BANGLORE |  
+----+-----+-----+
```

## 2. POJO class(EmployeeBo.java)

```
package bo;  
public class EmployeeBo {  
    private int eid;  
    private String name;  
    private String address;  
  
    //Setters& getters  
}
```

### 3. O/R Mapping XML(EmpLOYEEBo.hbm.xml)

```
<hibernate-mapping>
    <class name="bo.EmployeeBo" table="employee">
        <id name="eid" column="eid">
            <generator class="assigned" />
        </id>
        <property name="name" column="name" />
        <property name="address" column="address" />
    </class>
</hibernate-mapping>
```

### 4. Hibernate Configuration file

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory>
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/smlcodes</property>
<property name="hibernate.connection.username">root</property>
<property name="hibernate.connection.password">root</property>

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

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

## 1. SELECT Operation using Hibernate

We have following methods to perform SELECT Operation in Hibernate

```
Object get(Class , Serializable id) throws ObjectNotFoundException  
Object load(Class, Serializable id)
```

Parameter	get	load
<b>Database retrieval</b>	It always hits the database	It does not hit database
<b>Proxy</b>	It returns real object	It returns proxy object
<b>Id not found</b>	If it does not get the object with id, it returns null	If it does get the object with id, it throws ObjectNotFoundException
<b>Use</b>	If you are not sure if object with id exists or not, you can use get	If you are sure about existence of object, you can use load

## Example

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

        //1.Load Configuration
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

        //2.Create Session
        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        //3.Perform Operations
        Object ob = session.load(EmployeeBo.class, new Integer(1));
        EmployeeBo bo = (EmployeeBo) ob;

        System.out.println("SELECTED DATA\n =====");
        System.out.println("EID : "+bo.getEid());
        System.out.println("NAME : "+bo.getName());
        System.out.println("ADDRESS : "+bo.getAddress());
    }
}
```

```
log4j:WARN No appenders could be found for logger
org.hibernate.cfg.Environment).
log4j:WARN Please initialize the log4j system properly.
Hibernate: select employeebo0_.eid as eid0_0_, employeebo0_.name as name0_0_,
employeebo0_.address as address0_0_ from employee employeebo0_ where
employeebo0_.eid=?

SELECTED DATA
=====
EID : 1
NAME : Satya
ADDRESS : VIJAYAWADA
```

## Get vs load

```
public class App {
    public static void main(String[] args) {
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

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

        System.out.println("get : Database retrieval");
        Student bo1 = (Student) session.get(Student.class, new Integer(101));
        Student bo2 = (Student) session.get(Student.class, new Integer(101));

        System.out.println("get : ID not found.... ");
        Student bo3 = (Student) session.get(Student.class, new Integer(102));
        System.out.println(bo3);
```

```

        System.out.println("load : Database retrieval");
        Student bo4 = (Student) session.load(Student.class, new Integer(101));
        Student bo5 = (Student) session.load(Student.class, new Integer(101));

        System.out.println("load : ID not found.... ");
        Student bo6 = (Student) session.load(Student.class, new Integer(102));
        System.out.println(bo6);
    }

}

get : Database retrieval
Hibernate: select student0_.sno as sno0_0_, student0_.name as name0_0_,
student0_.address as address0_0_ from student student0_ where student0_.sno=?

Hibernate: select student0_.sno as sno0_0_, student0_.name as name0_0_,
student0_.address as address0_0_ from student student0_ where student0_.sno=?
//Two times it hits the Database

get : ID not found....
Hibernate: select student0_.sno as sno0_0_, student0_.name as name0_0_,
student0_.address as address0_0_ from student student0_ where student0_.sno=?
null
//No Error , it showing null

load : Database retrieval
//data with same id already loaded at get() call, it just returns Proxy Object

load : ID not found....
Hibernate: select student0_.sno as sno0_0_, student0_.name as name0_0_,
student0_.address as address0_0_ from student student0_ where student0_.sno=?
Exception in thread "main" org.hibernate.ObjectNotFoundException: No row with the
given identifier exists: [rewrite.Student#102]

```

## 2. INSERT Operation using Hibernate

We have Following methods to perform insert Operation in Hibernate

- Serializable **save(Object object)**
- void **persist(Object object)**
- void **saveOrUpdate(Object object)**

```

package curd;
public class EmployeeInsert {
    public static void main(String[] args) {
        // 1.Load Configuration
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

        // 2.Create Session
        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();
    }
}

```

```

// 3.Create Transaction Object
Transaction tx = session.beginTransaction();

EmployeeBo ob1 = new EmployeeBo(12, "KARTHIK", "ONGOLE");
session.save(ob1);
tx.commit();

tx.begin();
EmployeeBo ob3 = new EmployeeBo(10, "NAG", "HYD");
session.saveOrUpdate(ob3);
tx.commit();

EmployeeBo ob2 = new EmployeeBo(11, "PERSIST", "VIZAG");
session.persist(ob2);

System.out.println("Data Saved Successfully");
session.close();
sf.close();
}

}

```

```

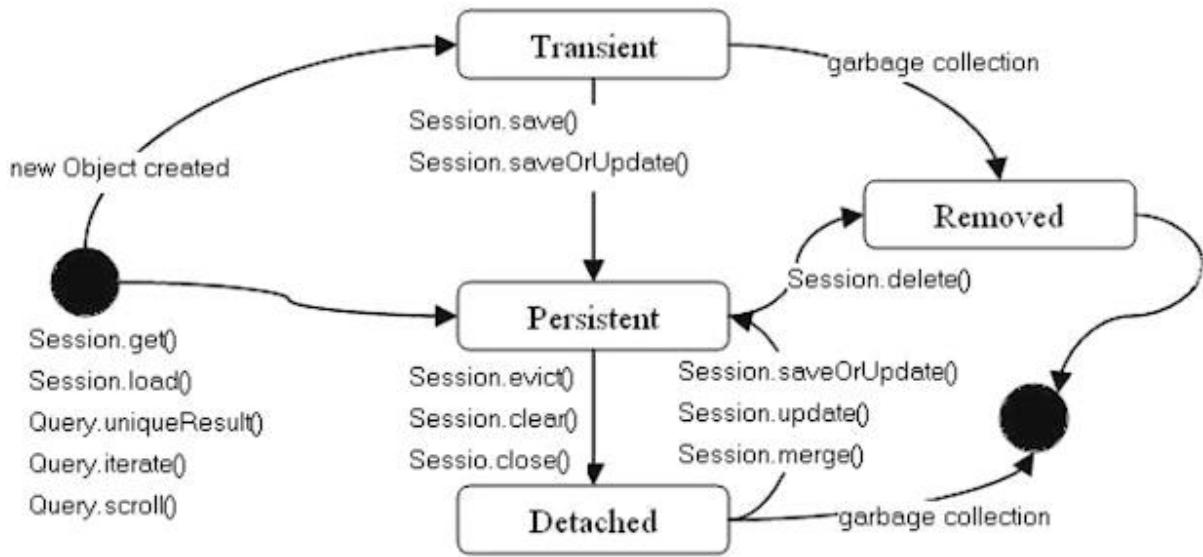
mysql> select * from employee;
+----+-----+-----+
| eid | name   | address |
+----+-----+-----+
| 1  | Satya  | VIJYAYAWADA |
| 2  | Ravi   | HYDERABAD  |
| 3  | SURYA  | HYDERABAD  |
| 4  | RAMAN  | PUNE        |
| 5  | DILEEP  | BANGLORE   |
| 6  | DILEEP  | BANGLORE   |
| 7  | ANANTH  | CHENNAI    |
| 8  | vijay   | CHENNAI    |
| 9  | KARTHIK | ONGOLE     |
| 10 | NAG    | HYD         |
| 12 | KARTHIK | ONGOLE     |
+----+-----+-----+

```

### Save vs Persist

- `save()` is Hibernate-proprietary, whereas `persist()` is a standard JPA method
- **`save()`** method can **return that primary key id** value which is generated by hibernate generator while inserting data, **`persist()` will never give any value back to the client**

```
long s = session.save(k);
```
- **`save()`** method will move the Student Object to **Transit state to Persistent state immediate**. But **`persist()`** method **doesn't guarantee** that the identifier value will be assigned to the persistent instance immediately, the assignment might happen **at flush time**.



- **At Transaction boundaries**, `persist()` method guarantees that it will not execute an INSERT statement if it is called outside of transaction boundaries. `save()` also doesn't insert but it will return the next generated key value.

```

public class App {
    public static void main(String[] args) {
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

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

        // Without Transaction it will return id if success
        System.out.println("save \n =====");
        Object o = session.save(new Student("Ravi", "HYD"));
        System.out.println("Next Generated Key value : " + o);

        // Without Transaction it wont commit transaction
        System.out.println("Persist \n =====");
        session.persist(new Student("Mohan", "HYD"));
    }
}

save
=====
Hibernate: select max(sno) from student
Next Generated Key value : 102
Persist
=====
```

`saveOrUpdate()` method either INSERT or UPDATE based upon existence of object in database. If persistence object already exists in database then UPDATE SQL will execute and if there is no corresponding object in database than INSERT will run.

### 3. UPDATE Operation using Hibernate

We have two approaches for updating already saved data in database.

#### Approach 1:

In this approach we load the existing row, and we will set the appropriate properties. On committing transaction hibernate automatically updates the data. But this is not recommended.

#### Approach 2:

In this approach to modify object in the database, we need to create new object with same id and we must call **update()** given by session interface

We have following method to perform UPDATE Operation in Hibernate

```
void update(Object object)
Object merge(Object object)
```

```
package curd;

public class EmployeeUpdate {

    public static void main(String[] args) {
        // 1.Load Configuration
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

        // 2.Create Session
        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        //=====Approach 1 ======
        Transaction tx = session.beginTransaction();
        EmployeeBo ob1 = (EmployeeBo) session.load(EmployeeBo.class, new Integer(4));
        ob1.setAddress("VIJAYAWADA");
        tx.commit();

        //=====Approach 2 ======
        tx = session.beginTransaction();
        EmployeeBo ob2 = new EmployeeBo(new Integer(5), "ANANTH", "HYDERABAD");
        session.update(ob2);
        tx.commit();
        System.out.println("Update Completed!");

        session.close();
        sf.close();

    }
}
```

```

mysql> select * from employee;
+---+---+---+
| eid | name | address |
+---+---+---+
| 1 | Satya | VIJAYAWADA |
| 2 | Ravi | HYDERABAD |
| 3 | SURYA | HYDERABAD |
| 4 | RAMAN | CHENNAI |
| 5 | ANANTH | CHENNAI |
+---+---+---+
5 rows in set (0.00 sec)

mysql> select * from employee;
+---+---+---+
| eid | name | address |
+---+---+---+
| 1 | Satya | VIJAYAWADA |
| 2 | Ravi | HYDERABAD |
| 3 | SURYA | HYDERABAD |
| 4 | RAMAN | VIJAYAWADA |
| 5 | ANANTH | HYDERABAD |
+---+---+---+
5 rows in set (0.00 sec)

```

Before update                          After Update

### What is the difference between and merge and update?

```

Employee emp1 = new Employee();
emp1.setEmpId(100);
emp1.setEmpName("Dinesh");
Session session1 = createNewHibernateSession();
session1.saveOrUpdate(emp1);
session1.close();
//emp1 object in detached state now

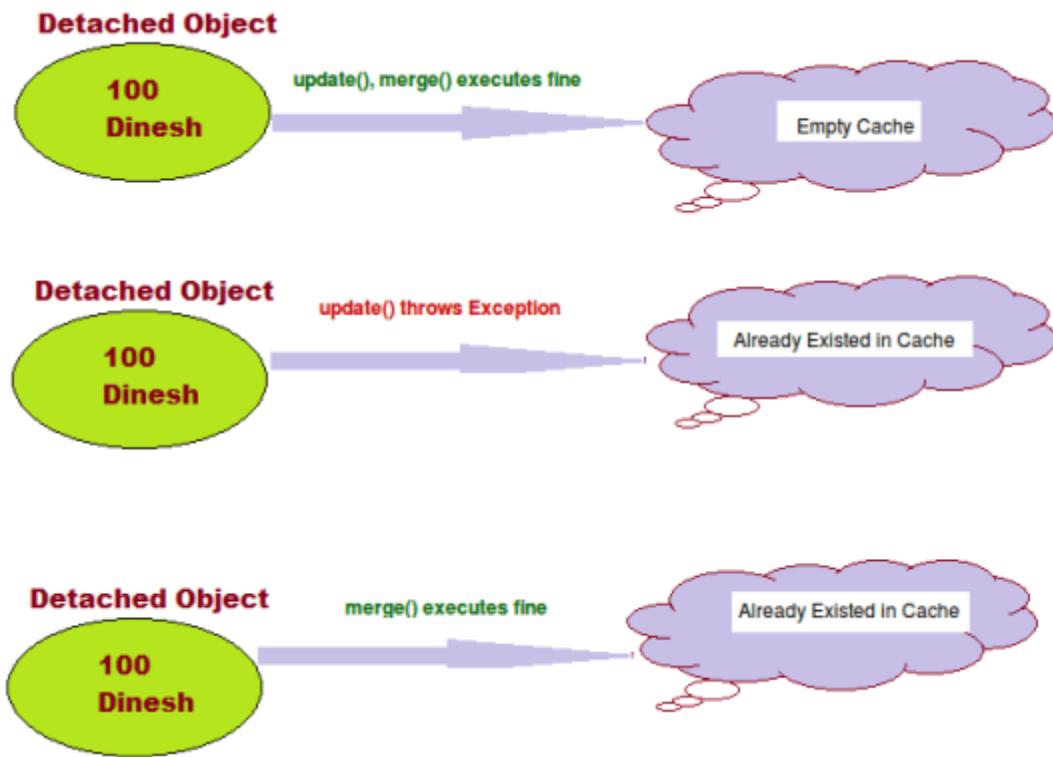
emp1.setEmpName("Dinesh Rajput");//Updated Name

//Create session again
Session session2 = createNewHibernateSession();
Employee emp2 =(Employee)session2.get(Employee.class, 100);
//emp2 object in persistent state with id 100

//below we try to make on detached object with id 100 to persistent state by using
update method of hibernate
    session2.update(emp1);//It occurs the exception NonUniqueObjectException
because emp2 object is having employee with same empid as 100. In order //to avoid
this exception we are using merge like given below instead of session.update(emp1);

session2.merge(emp1); //it merge the object state with emp2
session2.update(emp1); //Now it will work with exception

```



#### Update:

- Suppose we are dealing with any employee object in the same session then we should use update() or saveOrUpdate() method.
- if you are sure that the session does not contain an already persistent instance with the same identifier, then use update to save the data in hibernate

#### Merge:

- Suppose we are creating a session and load an employee object. Now object in session cache. If we close the session at this point and we edit state of object and tried to save using update() it will throw exception. To make object persistent we need to open another session. Now we load same object again in current session. So if we want to update present object with previous object changes we have to use merge() method. Merge method will merge changes of both states of object and will save in database.
- if you want to save your modifications at any time without knowing about the state of a session, then use merge() in hibernate.

## 4. DELETE Operation using Hibernate

We have following method to perform DELETE Operation in Hibernate

```
void delete(Object object)
```

```
package curd;
import bo.EmployeeBo;

public class EmployeeDelete {

    public static void main(String[] args) {
        // 1.Load Configuration
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

        // 2.Create Session
        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        //=====Load the Object & DELETE ======
        Transaction tx = session.beginTransaction();
        EmployeeBo bo =(EmployeeBo) session.load(EmployeeBo.class, new
Integer(4));
        session.delete(bo);
        tx.commit();

        session.close();
        sf.close();

    }
}
```

```
mysql> select * from employee;
+---+-----+-----+
| eid | name | address |
+---+-----+-----+
| 1 | Satya | VIJAYAWADA |
| 2 | Ravi | HYDERABAD |
| 3 | SURYA | HYDERABAD |
| 4 | RAMAN | VIJAYAWADA |
| 5 | ANANTH | HYDERABAD |
+---+-----+-----+
5 rows in set (0.00 sec)
```

Before Delete

```
mysql> select * from employee;
+---+-----+-----+
| eid | name | address |
+---+-----+-----+
| 1 | Satya | VIJAYAWADA |
| 2 | Ravi | HYDERABAD |
| 3 | SURYA | HYDERABAD |
| 5 | ANANTH | HYDERABAD |
+---+-----+-----+
4 rows in set (0.00 sec)
```

After DELETE

## 5. Clear

### Clear

```
void evict(Object object) : Remove this instance from the session cache.
void clear() : Completely clear the session.
Void close() : close the session
```

### Example

```
SQL> SELECT * FROM EMPLOYEE;
   EMPNO  EMPNAME          EMPDEPT
-----  -----
    2 Sree               R&D
    1 Kalyan             Developement
```

**Clear () :** When this method get called inside transaction boundry then all objects which are currently associate with particular session will be disconnected / clean or no longer associate with that Session instance.

```
Employee emp1 = (Employee) session.get(Employee.class, 1);
Employee emp2 = (Employee) session.get(Employee.class, 2);
// emp1 and emp2 are in persistent state.

emp1.setEmpName("NEW SREE");
emp2.setEmpName("NEW KALYAN");

session.clear();
// emp1 and emp2 are in detached state.

session.getTransaction().commit();
session.close();

EMPNO  EMPNAME          EMPDEPT
-----  -----
    2 Sree               R&D
    1 Kalyan             Developement
```

Therefore, after calling this method nothing will be performed on persistence layer or DB.

**evict():** Removes the object from the session. This method is used to dissociate/disconnect the specified object from the session.

```
Employee emp1 = (Employee) session.get(Employee.class, 1);
Employee emp2 = (Employee) session.get(Employee.class, 2);
//emp1 and emp2 are in persistent state.

emp1.setEmpName("NEW KALYAN"); // it will removed
emp2.setEmpName("NEW SREE");

session.evict(emp1);
//emp1 is in detached state and emp2 is in persistent state.

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

```
EMPNO  EMPNAME          EMPDEPT
-----  -----
    2 NEW SREE           R&D
    1 Kalyan             Developement
```

**Close() :** Close session by calling **session.close()** method, means End the session and releasing the JDBC Connection and clean up.

## 4. Hibernate POJO Class Lifecycle

In Hibernate POJO class (Persistance class) object will have 3 States

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

### 1. Transient state

- Whenever an object of a pojo class is created then it will be in the **Transient state**
- When the object is in a Transient state it **doesn't represent any row of the database**
- If we modify the data of a pojo class object, when it is in transient state then **it doesn't effect on the database table**

### 2. Persistent state

- When the object is in persistent state, then it represents one row of the database
- if the object is in persistent state then it is associated with the **unique Session**

### 3. Detached state

- After Persistent State Object will goes under Dethatched State
- if we want to move an object from persistent to detached state, we need to do either **closing that session** or need to **clear the cache of the session**

```
package curd;

public class POJOLifeCycle {

    public static void main(String[] args) {
        // 1.Load Configuration
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

        // 2.Create Session
        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        //=====1.Transient State START=====
        EmployeeBo bo = new EmployeeBo();
        bo.setEid(6);
        bo.setName("RAJESH");
        bo.setAddress("NEWYORK");
        //=====1.Transient State END=====

        //=====2.Persistent state START=====
    }
}
```

```

        Transaction tx = session.beginTransaction();
        session.save(bo);
        tx.commit();
        //=====2.Persistent state END=====

        //=====3.Detached State START=====
        session.close();
        bo.setEid(7);
        bo.setName("MADHU");
        bo.setAddress("COLOMBO");
        //=====3.Detached State END=====
        sf.close();

    }

}

```

## 5. Hibernate Inheritance Mapping

---

In hibernate inheritance, if we have base and derived classes, now **if we save derived(sub) class object, base class object will also be stored into the database.**

Hibernate supports 3 types of Inheritance Mappings:

1. **Table per class hierarchy**
2. **Table per sub-class hierarchy**
3. **Table per concrete class hierarchy**

We will understand one by one with examples. Below tables are used in upcoming examples.

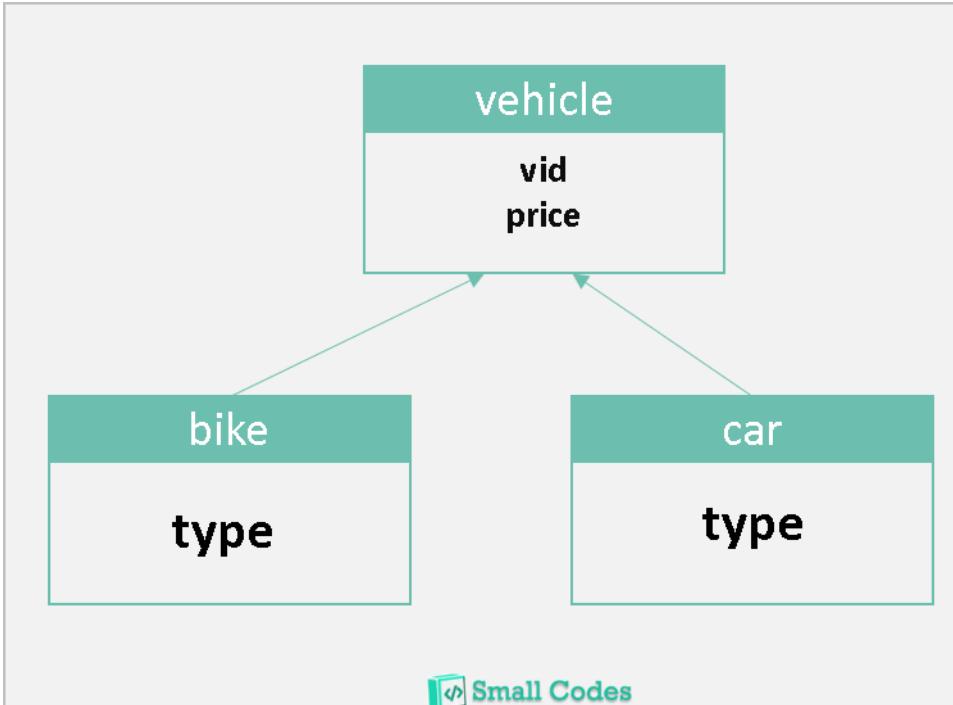
```

CREATE TABLE `vehicle` (
`vid` INT(11) NOT NULL AUTO_INCREMENT,
`price` DOUBLE NOT NULL DEFAULT '0',
PRIMARY KEY (`vid`));

CREATE TABLE `bike` (
`type` VARCHAR(50) NULL DEFAULT NULL
);

CREATE TABLE `car` (
`type` VARCHAR(50) NULL DEFAULT NULL
);

```



Following files are common to all examples, only we need to change mapping xml & Application class

### Vehicle.java

```

package inheritance;

public class Vehicle {
    private int vid;
    private double price;

    public int getVid() {
        return vid;
    }

    public void setVid(int vid) {
        this.vid = vid;
    }

    public double getPrice() {
        return price;
    }

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

### Bike.java

```
package inheritance;

public class Bike extends Vehicle {
    private String biketype;

    public String getBiketype() {
        return biketype;
    }

    public void setBiketype(String biketype) {
        this.biketype = biketype;
    }
}
```

### Car.java

```
package inheritance;
public class Car extends Vehicle {
    private String cartype;

    public String getCartype() {
        return cartype;
    }

    public void setCartype(String cartype) {
        this.cartype = cartype;
    }
}
```

### hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/smlcodes</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>

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

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

### InheritanceCommonApp.java

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

public class TablePerClassExample {

    public static void main(String[] args) {
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

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

        Bike bike = new Bike();
        bike.setVid(101);
        bike.setBiketype("HONDA");
        bike.setPrice(50000);

        Car car = new Car();
        car.setVid(102);
        car.setCartype("MARUTHI");
        car.setPrice(600000);

        Transaction tx = session.beginTransaction();
        session.save(bike);
        session.save(car);
        tx.commit();
        session.close();
        factory.close();
    }
}
```

## 1. Table per class hierarchy

If we **save** the derived class object in the database, then automatically base class data will also be saved into the database in base class Table

For example, if we save the **derived class** object like Car or Bike then automatically Vehicle class object will also be saved into the database, and in the **database** all the data will be stored into a **single table** only, which is base class table.

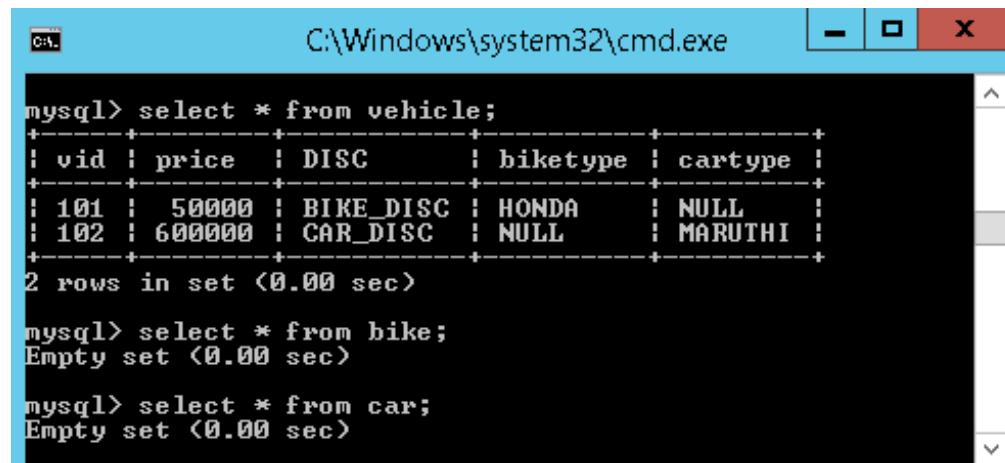
For this type of hierarchy, we must use one extra **discriminator column** in the database, to identify which **derived** class object we have been saved in the table along with the base class object, if we are not using this column hibernate will **throws the exception**.

### Vehicle.hbm.xml

```
<hibernate-mapping>
  <class name="inheritance.Vehicle" table="vehicle">
    <id name="vid" column="vid"></id>
    <discriminator column="DISC" type="string"/>
    <property name="price" column="price"></property>

    <subclass name="inheritance.Bike" discriminator-value="BIKE_DISC">
      <property name="biketype" column="biketype"></property>
    </subclass>

    <subclass name="inheritance.Car" discriminator-value="CAR_DISC">
      <property name="cartype" column="cartype"></property>
    </subclass>
  </class>
</hibernate-mapping>
```



The screenshot shows a Windows Command Prompt window titled 'cmd.exe' with the path 'C:\Windows\system32\'. The window displays MySQL command-line interface output:

```
mysql> select * from vehicle;
+----+-----+-----+-----+-----+
| vid | price | DISC | biketype | cartype |
+----+-----+-----+-----+-----+
| 101 | 50000 | BIKE_DISC | HONDA | NULL      |
| 102 | 60000 | CAR_DISC   | NULL   | MARUTHI   |
+----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from bike;
Empty set (0.00 sec)

mysql> select * from car;
Empty set (0.00 sec)
```

We added one new line discriminator, after the id element just to identify which derived class object we have been saved in the table. Here everything has been saved in a single table(vehicle)

## 2. Table per sub-class hierarchy

In this type of hierarchy, **if we save the Base class object, first hibernate will saves the Base class object into the base class table, then it will save the subclass object data into subclass table.** Here first it will save data into Base class table.

In below Example, if we save the Car/Bike class object, first hibernate will saves the data related to Vehicle class object into the vehicle table and then Car/Bike object data in Car/Bike related tables.so we can say, **No. of classes equals to No. of Tables**

**No. of classes = No. of Tables**

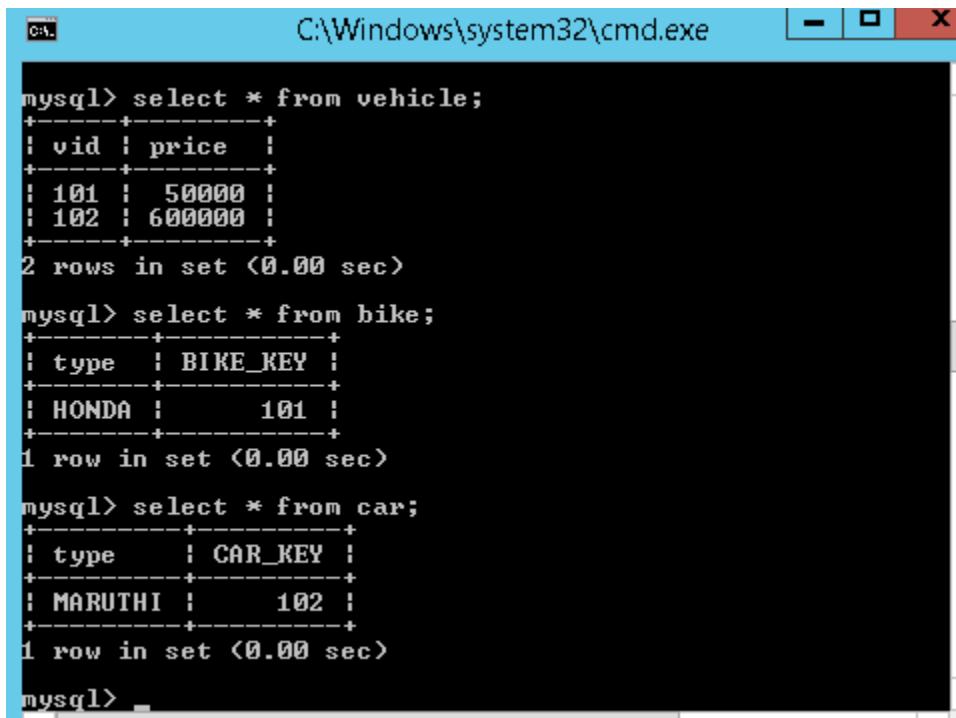
Here **<joined-subclass>** element of **<class>** is used to map the child class with parent class using the primary key and foreign key relation

### Vehicle.hbm.xml

```
<hibernate-mapping>
    <class name="inheritance.Vehicle" table="vehicle">
        <id name="vid" column="vid"></id>
        <property name="price" column="price"></property>

        <joined-subclass name="inheritance.Bike" table="bike">
            <key column="BIKE_KEY" />
            <property name="biketype" column="type"></property>
        </joined-subclass>

        <joined-subclass name="inheritance.Car" table="car">
            <key column="CAR_KEY" />
            <property name="cartype" column="type"></property>
        </joined-subclass>
    </class>
</hibernate-mapping>
```



The screenshot shows a Windows Command Prompt window titled 'cmd' with the path 'C:\Windows\system32\cmd.exe'. The window contains the following MySQL session:

```
mysql> select * from vehicle;
+---+---+
| vid | price |
+---+---+
| 101 | 50000 |
| 102 | 600000 |
+---+---+
2 rows in set (0.00 sec)

mysql> select * from bike;
+---+---+
| type | BIKE_KEY |
+---+---+
| HONDA |      101 |
+---+---+
1 row in set (0.00 sec)

mysql> select * from car;
+---+---+
| type | CAR_KEY |
+---+---+
| MARUTHI |     102 |
+---+---+
1 row in set (0.00 sec)

mysql> _
```

- Once we save the **derived** class object, then hibernate will first save the **base class** object then derived class object.
- At the time of saving the derived class object, hibernate will copy the **primary key** value of the base class into the corresponding derived class, by using **<key>** tag. From the above output
  - 102 copied into **CAR\_KEY** column of **car** table
  - 101 copied into **BIKE\_KEY** column of the **bike** table

### 3. Table per concrete class hierarchy

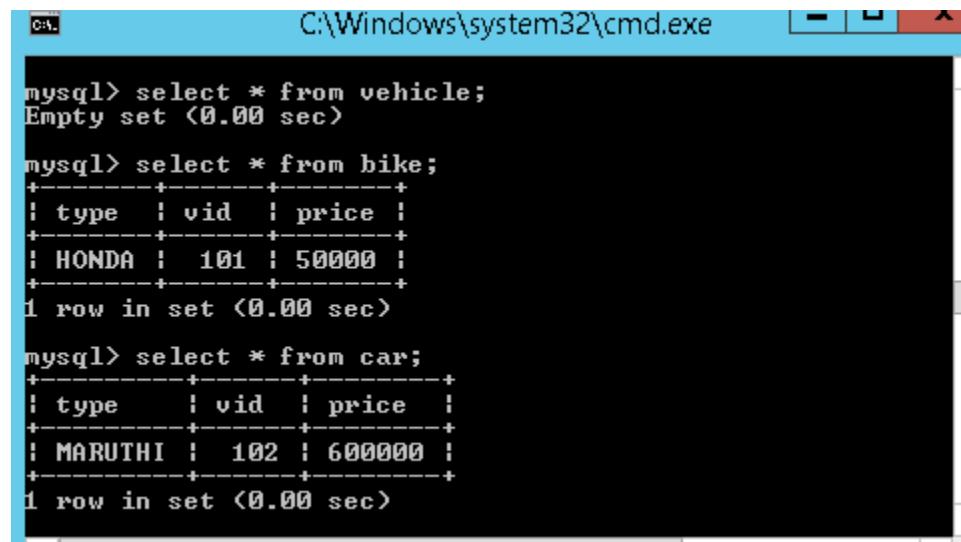
- Once we save the **derived class object**, then **derived class data and base class data will be saved in the derived class related table** in the database
- for this type we **need** the tables for **derived classes**, but **not for the base class**
- we need to use one new element **<union-subclass>**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

#### Vehicle.hbm.xml

```
<hibernate-mapping>
    <class name="inheritance.Vehicle" table="vehicle">
        <id name="vid" column="vid"></id>
        <property name="price" column="price"></property>

        <union-subclass name="inheritance.Bike" table="bike">
            <property name="biketype" column="type"></property>
        </union-subclass>

        <union-subclass name="inheritance.Car" table="car">
            <property name="cartype" column="type"></property>
        </union-subclass>
    </class>
</hibernate-mapping>
```



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. It contains the following MySQL session:

```
mysql> select * from vehicle;
Empty set (0.00 sec)

mysql> select * from bike;
+-----+-----+-----+
| type | vid  | price |
+-----+-----+-----+
| HONDA | 101  | 50000 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from car;
+-----+-----+-----+
| type   | vid  | price  |
+-----+-----+-----+
| MARUTHI | 102  | 600000 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

## 6. Hibernate Generators <generator>

The <generator> subelement of id used to **generate the unique identifier** for the objects of persistent class. There are many generator classes defined in the Hibernate Framework.

### Example:

```
<hibernate-mapping>
    <class name="bo.EmployeeBo" table="employee">
        <id name="eid" column="eid">
            <generator class="assigned" />
        </id>
        <property name="name" column="name" />
        <property name="address" column="address" />
    </class>
</hibernate-mapping>
```

In Above example `class="assigned"` means, while inserting data into database user will take care about generating Primary key value. But, hibernate can also generate the primary keys without user interaction, by using **Generators**.

### List of generators

- hibernate using different primary key generator algorithms, for each algorithm internally a class is created by hibernate for its implementation
- hibernate provided different primary key generator classes and all these classes are implemented from `org.hibernate.id.IdentifierGenerator` Interface

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

1. **assigned**
2. **increment**
3. **sequence**
4. **identity**
5. **hilo**
6. **native**
7. **foreign**
8. **uuid**

## 1.assigned

- This is the **default generator class** used by the hibernate, if we do not specify `<generator>` element under id element then hibernate by **default assumes it as "assigned"**
- If generator class is assigned, then the **programmer is responsible for assigning the primary key** value to object which is going to save into the database

```
<class name="bo.EmployeeBo" table="employee">
    <id name="eid" column="eid">
        <generator class="assigned" />
    </id>
    <property>....</property>
</class>
```

## 2.increment

- First select the max id if there, if no 1 as max
- for each record it **increments by 1 (i++)**
- Increment will take **care by Application Layer[Hibernate]**

```
<class name="bo.EmployeeBo" table="employee">
    <id name="eid" column="eid">
        <generator class="increment"/>
    </id>
    <property>....</property>
</class>
```

## 3.identity

- First select the max id if there, if no 1 as max
- for each record it **increments by 1 (i++)**
- Increment will **have taken care by DB Layer[MySQL]**
- MySQL, DB2 Support this. Oracle Won't Support it.

```
<class name="bo.EmployeeBo" table="employee">
    <id name="eid" column="eid">
        <generator class=" identity"/>
    </id>
    <property>....</property>
</class>
```

## 4.sequence

- while inserting a new record in a database, **hibernate gets next value from the sequence** under assigns that value for the new record
- If programmer has created a sequence in the database, then that sequence name should be passed as the generator
- **Sequence start with 1 and Incremented by 1**
- Internally it creates Sequence table, and increment operations done here
- **Increment will take care by Both Application Layer[Hibernate] & DB Layer[MySQL]**
- MySQL, DB2 Support this. Oracle Won't

```
<class name="bo.EmployeeBo" table="employee">
    <id name="eid" column="eid">
        <generator class="sequence"/>
    </id>
    <property>....</property>
</class>
```

For defining your own sequence, use the param subelement of generator.

```
<class name="bo.EmployeeBo" table="employee">
    <id name="eid" column="eid">
        <param name="sequence">USER_DEFINED_SEQUENCE</param>
    </id>
    <property>....</property>
</class>
```

## 5.hilo

- hilo **start with 0**
- Internally it creates **hilo table**
- **It will take High value form Hilo table, and it will Increment**
- for each Deployment/Restart application, **High value increment by 32768**
- 1 st Deploy: 1,2,3,4...
- 2 nd Deploy: 32768,32769.....
- MySQL, DB2 Support this. Oracle Won't Support it.

```
<class name="bo.EmployeeBo" table="employee">
    <id name="eid" column="eid">
        <generator class="hilo"/>
    </id>
    <property>....</property>
</class>
```

## 6.native

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

- **identity**
- **sequence**
- **hilo**

```
<class name="bo.EmployeeBo" table="employee">
    <id name="eid" column="eid">
        <generator class="native"/>
    </id>
    <property>....</property>
</class>
```

## 7.foregin

It uses the id of another associated object, mostly used with <one-to-one> association.

## 8.uuid

It uses 128-bit UUID algorithm to generate the id. The returned id is of type String, unique within a network (because IP is used). The UUID is represented in hexadecimal digits, 32 in length.

## Custom generator

For ur application, you want to generate keys as per ur wish like **icici\_101, icici\_102, icici\_103, .....**

- write **IciciGenerator** class implementing **IdentityGenerator**
- override generate method
- and write logic for keys

```
<class name="bo.EmployeeBo" table="employee">
    <id name="eid" column="eid">
        <generator class="IciciGenerator"/>
    </id>
    <property>....</property>
</class>
```

## Generators Example

### EmployeeBo.java

```
package bo;
public class EmployeeBo {
    private int eid;
    private String name;
    private String address;
    //Setters & Getters
}
```

### GeneratorsExample.java

```
package app;

import org.hibernate.*;
import org.hibernate.cfg.*;
import bo.EmployeeBo;

public class GeneratorsExample {

    public static void main(String[] args) {
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

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

        EmployeeBo bo = new EmployeeBo();
        //bo.setEid(100);
        bo.setName("sequence");
        bo.setAddress("BANGLORE");

        Transaction tx = session.beginTransaction();
        session.save(bo);

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

### EmployeeBo.hbm.xml

```
<hibernate-mapping>
    <class name="bo.EmployeeBo" table="employee">
        <id name="eid" column="eid">
            <generator class="sequence" />
        </id>
        <property name="name" column="name" />
        <property name="address" column="address" />
    </class>
</hibernate-mapping>
```

Here we are changing generator classes one-by-one, check the how data stored in DB in output window.

### hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"hibernate-configuration-3.0.dtd">

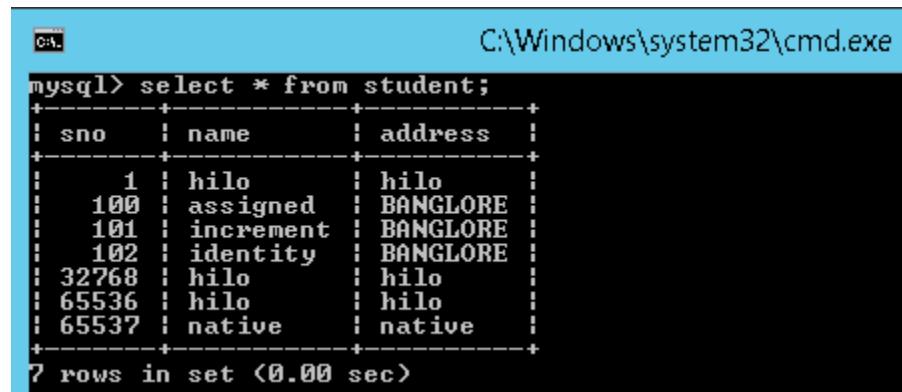
<hibernate-configuration>
    <session-factory>
        <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/smlcodes</property>
            <property name="hibernate.connection.username">root</property>
            <property name="hibernate.connection.password">root</property>

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

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

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

### Output



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. Inside the window, a MySQL command is run: 'mysql> select \* from student;'. The output displays a table with columns 'sno', 'name', and 'address'. The data rows are:

sno	name	address
1	hilo	hilo
100	assigned	BANGLORE
101	increment	BANGLORE
102	identity	BANGLORE
32768	hilo	hilo
65536	hilo	hilo
65537	native	native

At the bottom of the output, it says '7 rows in set (0.00 sec)'

For Sequence,  
Exception in thread "main" [org.hibernate.MappingException](#): could not instantiate id generator  
at  
[org.hibernate.id.IdentifierGeneratorFactory.create\(IdentifierGeneratorFactory.java:98\)](#)

## 7. Hibernate Query Language (HQL)

**Hibernate Query Language (HQL)** is same as SQL (Structured Query Language) but it doesn't depend on the table of the database. Instead of table name, we use class name in HQL. So it is database independent query language

- **HQL is database independent**, means if we write any program using HQL commands then our program will be able to execute in all the databases without doing any further changes to it.
- HQL supports object oriented features like **Inheritance, polymorphism, Associations** (Relationships)

### SQL vs HQL

If we want to select a **Complete Object** from the database, we use POJO class reference in place of \* while constructing the query

```
// In SQL
sql> select * from Employee
Note: Employee is the table name.

// In HQL
hql> select s from EmployeeBo s
[ or ]
from EmployeeBo s
Note: here s is the reference of EmployeeBo
```

If we want to load the **Partial Object** from the database that is only selective properties of an objects, then we need to replace column names with POJO class variable names

```
// In SQL
sql> select eid,name,address from Employee
Note: eid,name,address are the columns of Employee the table.

// In HQL
hql> select s.eid,s.name,s.address from EmployeeBo s
```

It is also possible to **select** the object from the database **by passing run time values** into the query using "?"

```
//In SQL
sql> select * from employee where eid=?

// In HQL
hql> select s from EmployeeBo s where s.eid=?
[ or ]
    select s from EmployeeBo s where s.eid =:101
```

## Query Interface

If we want to execute a HQL query on a database, we need to create a **Query** interface object. To get query object, we need to call **session.createQuery()** method in the session Interface.

Following are the most commonly used methods in Query interface

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

### Syntax:

```
Query qry = session.createQuery("---- HQL command ---");
List l = qry.list();
Iterator it = l.iterator();
while(it.hasNext())
{
    Object o = it.next();
    EmployeeBo s = (EmployeeBo)o;
    -----
}
```

## HQL Examples

Following files are common to all the examples

### Table :bank

```
CREATE TABLE `bank` (
`accno` INT(11) NOT NULL AUTO_INCREMENT,
`accname` VARCHAR(50) NULL DEFAULT NULL,
`balance` DOUBLE NULL DEFAULT NULL,
PRIMARY KEY (`accno`)
)
ENGINE=InnoDB
;
```

```
mysql> select * from bank;
+-----+-----+-----+
| accno | accname | balance |
+-----+-----+-----+
|   121 | Satya  | 589000  |
|   127 | Surya   | 56000   |
|   133 | Ravi    | 800000  |
+-----+-----+-----+
```

### BankBo.java

```
package hql;

public class BankBo {
    private int accno;
    private String accname;
    private double balance;

    public int getAccno() {
        return accno;
    }
    public void setAccno(int accno) {
        this.accno = accno;
    }
    public String getAccname() {
        return accname;
    }
    public void setAccname(String accname) {
        this.accname = accname;
    }
    public double getBalance() {
        return balance;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }
}
```

### BankBo.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM "hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="hql.BankBo" table="bank">
        <id name="accno" column="accno" />
        <property name="accname" column="accname" />
        <property name="balance" column="balance" />
    </class>
</hibernate-mapping>
```

### hibernate.cfg.xml

```
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/smlcodes</property>
            <property name="hibernate.connection.username">root</property>
            <property name="hibernate.connection.password">root</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
            <mapping resource="BankBo.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

## 1. HQL Select Query Example

```
package hql;

import java.util.Iterator;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.classic.Session;

public class HQLSelect {
    public static void main(String[] args) {
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

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

        System.out.println("====1.Selecting Complete Object===== ");
        Query query = session.createQuery("Select b from BankBo b");
        List list = query.list();
        Iterator it = list.iterator();
        while (it.hasNext()) {
            BankBo bo = (BankBo) it.next();
            System.out.println("-----");
            System.out.println("ACC No : " + bo.getAccno());
            System.out.println("Name : " + bo.getAccname());
            System.out.println("Balance : " + bo.getBalance());
        }

        System.out.println("====2.Selecting Partial Object===== ");
        query = session.createQuery("Select b.accname, b.balance from
BankBo b where b.accno=?");
        query.setParameter(0, new Integer(127));
        list = query.list();
        it = list.iterator();
        while (it.hasNext()) {
            Object o[] = (Object[]) it.next();
            System.out.println("Name : " + o[0]);
            System.out.println("Balance : " + o[1]);
        }
    }
}
```

```
====1.Selecting Complete Object=====
ACC No : 121
Name : Satya
Balance : 589000.0
-----
ACC No : 127
Name : Surya
Balance : 56000.0
====2.Selecting Partial Object=====
Name : Surya
Balance : 56000.0
```

## 2. HQL Update/Delete Query Example

while working with DML operations in HQL we have to call **executeUpdate()**; to execute the query, which **will returns one integer value** after the execution it will **tells the count of effected rows**.

```
package hql;

import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.classic.Session;

public class HQLUpdateDelete {
    public static void main(String[] args) {
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

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

        System.out.println("===== UPDATE OPERATION =====");
        Transaction tx = session.beginTransaction();
        Query query = session.createQuery("update BankBo b set b.balance =? where b.accno=?");
        query.setParameter(0, new Double(0));
        query.setParameter(1, new Integer(127));
        int rs = query.executeUpdate();
        tx.commit();
        System.out.println(rs + " :Rows are Updated");

        System.out.println("===== DELETE OPERATION =====");
        tx = session.beginTransaction();
        query = session.createQuery("delete from BankBo b where b.accno=?");
        query.setParameter(0, new Integer(133));
        rs = query.executeUpdate();
        System.out.println(rs + " :Rows are Updated");
        tx.commit();
        session.close();
    }
}
```

```
mysql> select * from bank;
+-----+-----+
| accno | accname | balance |
+-----+-----+
| 121  | Satya   | 589000  |
| 127  | Surya   |      0  |
+-----+-----+
2 rows in set (0.00 sec)
```

### **3. HQL Insert Query Example**

while writing the insert query, we need to select values from other table, we can't insert our own values manually. Because, HQL supports only the **INSERT INTO..... SELECT.....** ; it won't support **INSERT INTO.....VALUES**

```
Query query = session.createQuery("insert into Stock(stock_code, stock_name)" +  
    "select stock_code, stock_name from backup_stock");  
int result = query.executeUpdate();
```

### **HQL with Aggregate functions**

We may call avg(), min(), max() etc. aggregate functions by HQL. Let's see some common examples:

#### **Example to get total salary of all the employees**

```
Query q=session.createQuery("select sum(salary) from Emp");  
List<Integer> list=q.list();  
System.out.println(list.get(0));
```

#### **Example to get maximum salary of employee**

```
Query q=session.createQuery("select max(salary) from Emp");
```

#### **Example to get minimum salary of employee**

```
Query q=session.createQuery("select min(salary) from Emp");
```

#### **Example to count total number of employee ID**

```
Query q=session.createQuery("select count(id) from Emp");
```

#### **Example to get average salary of each employees**

```
Query q=session.createQuery("select avg(salary) from Emp");
```

## 8. Hibernate Criteria Query Language(HCQL)

Criteria is nothing but a **Condition**. CQL is used to apply conditions on **SELECT** Queries. The **Criteria interface** provides methods to apply criteria on SELECT queries.

### Criteria Interface

The Criteria interface provides many methods to specify criteria. The object of Criteria can be obtained by calling the **createCriteria()** method of Session interface

```
public Criteria createCriteria(Class c)
```

commonly used methods of Criteria interface are as follows:

1. **public Criteria add(Criterion c)** is used to add restrictions.
2. **public Criteria addOrder(Order o)** specifies ordering.
3. **public Criteria setFirstResult(int firstResult)** specifies the first number of record to be retrieved.
4. **public Criteria setMaxResult(int totalResult)** specifies the total number of records to be retrieved.
5. **public List list()** returns list containing object.
6. **public Criteria setProjection(Projection projection)** specifies the projection

### Working with Criteria Query Language

#### Example Criteria

```
Criteria crit = session.createCriteria(Products.class);
Criterion c1=Restrictions.gt("price", new Integer(12000));
//price is our pojo class variable
crit.add(c1); // adding criterion object to criteria class object
List l = crit.list(); // executing criteria query
```

- If we want to put, we need to create one **Criterion Interface** object and we need to **add()** this object to **Criteria Class object**
- In order to get Criterion object, we need to use **Restrictions class**. Restrictions is the factory for producing Criterion objects
- In **Restrictions class, we have all static methods** and each method of this class **returns Criterion object**

**REMEMBER: CQL is only used on SELECT Queries**

## Example: Get the details from bank table where balance>3000.

Table : Bank

```
mysql> select * from bank;
+-----+-----+-----+
| accno | accname | balance |
+-----+-----+-----+
| 101   | Satya   |    2000 |
| 102   | Surya   |    3000 |
| 103   | Ravi    |    1000 |
| 104   | Rakesh  |    4000 |
| 105   | CHANDU  |    5000 |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

### BankBo.java

```
package hql;

public class BankBo {
    private int accno;
    private String accname;
    private double balance;

    public int getAccno() {
        return accno;
    }
    public void setAccno(int accno) {
        this.accno = accno;
    }
    public String getAccname() {
        return accname;
    }
    public void setAccname(String accname) {
        this.accname = accname;
    }
    public double getBalance() {
        return balance;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }
}
```

### BankBo.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping SYSTEM "hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="hql.BankBo" table="bank">
        <id name="accno" column="accno" />
        <property name="accname" column="accname" />
        <property name="balance" column="balance" />
    </class>
</hibernate-mapping>
```

### hibernate.cfg.xml

```
<hibernate-configuration>
    <session-factory>
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/smlcodes</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>
<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

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

```
package cql;

import java.util.Iterator;
import java.util.List;
import org.hibernate.*;
import hql.BankBo;

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

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

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

        Criteria criteria = session.createCriteria(BankBo.class);
        Criterion cn = Restrictions.gt("balance", new Double(3000));
        criteria.add(cn);
        List list = criteria.list();

        System.out.println("List Size : " + list.size());
        Iterator it = list.iterator();

        while (it.hasNext()) {
            BankBo bo = (BankBo) it.next();
            System.out.println(bo.getAccno() + ", " + bo.getAccname() + ", "
+ bo.getBalance());
        }
        session.close();
        factory.close();
    }
}
```

```
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.Environment).
log4j:WARN Please initialize the log4j system properly.
Hibernate: select this_.accno as accno0_0_, this_.accname as accname0_0_,
this_.balance as balance0_0_ from bank this_ where this_.balance>?
List Size : 2
104, Rakesh, 4000.0
105, CHANDU, 5000.0
```

## Example: Adding ORDERBY Conditions to Criteria

If we want to add some sorting order for the objects, we need to add an **Order class** object to the Criteria class object by calling **addOrder() method**. In Order class, we have **asc()** and **dsc()** for getting an objects in required order.

```
public class CriteriaOrderExample {  
    public static void main(String[] args) {  
  
        Configuration cfg = new Configuration();  
        cfg.configure("hibernate.cfg.xml");  
  
        SessionFactory factory = cfg.buildSessionFactory();  
        Session session = factory.openSession();  
  
        Criteria criteria = session.createCriteria(BankBo.class);  
        Criterion cn = Restrictions.gt("balance", new Double(2000));  
        criteria.add(cn);  
        criteria.addOrder(Order.asc("balance"));  
        List list = criteria.list();  
        Iterator it = list.iterator();  
  
        while (it.hasNext()) {  
            BankBo bo = (BankBo) it.next();  
            System.out.println(bo.getAccno() + ", " + bo.getBalance());  
        }  
        session.close();  
        factory.close();  
    }  
}
```

```
log4j:WARN No appenders could be found for logger  
102, 3000.0  
104, 4000.0  
105, 5000.0
```

hibernate will select the records (rows) from **BankBo** table and stores them into a **ResultSet** and then converts each row data of resultset into a **POJO class object** basing on our field type, then all these objects into a list according to the order you have given

## Example: Adding multiple conditions

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

```
crit.add(Restrictions.and(Restrictions.like("accname", "%satya%"),  
    Restrictions.eq("price", new Integer(12000))));  
List l=crit.list();  
Iterator it = l.iterator();
```

Like this we can add any number of conditions

## Hibernate Projections

In criteria, we are able to load complete object only, to load the partial objects (Selected Columns only) we need to use projections.

- **Projection** is an Interface, **Projections** is a class for producing projection objects.
- In **Projections** class, we have **all static methods** and each method of this class returns Projection interface object.
- If we want to add a Projection object to Criteria then we need to call a method **setProjection()**

## Projections Syntax

while adding projection object to criteria, it is possible to **add one object at a time**. It means if we add 2nd projection object then this 2nd one will override the first one (first one won't work), so at a time we can only one projection object to criteria object.

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

### Example:

```
Criteria crit = session.createCriteria(Products.class);
crit.setProjection(Projections.property("proName"));
List l=crit.list();
Iterator it=l.iterator();
while(it.hasNext())
{
    String s = (String)it.next();
    // ---- print -----
}
```

## Example 1: Load Single Column using Projections

In below example we are using same configuration files and BankBo.java as above examples

```
public class ProjectionsDemo {
    public static void main(String[] args) {
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");
        SessionFactory factory = cfg.buildSessionFactory();
        Session session = factory.openSession();
        Criteria criteria = session.createCriteria(BankBo.class);
        Criterion cn = Restrictions.gt("balance", new Double(2000));
        criteria.add(cn);

        criteria.setProjection(Projections.property("balance"));
        List list = criteria.list();

        System.out.println("List Size : " + list.size());
    }
}
```

```

        Iterator it = list.iterator();
        while (it.hasNext()) {
            Double bal = (Double) it.next();
            System.out.println("Balance : " + bal);
        }
        session.close();
        factory.close();
    }
}

```

```

Hibernate: select this_.balance as y0_ from bank this_ where this_.balance>?
List Size : 3
Balance : 3000.0
Balance : 4000.0
Balance : 5000.0

```

## Example 2: Load Multiple Columns using Projections

If we want to **load partial object, with multiple columns** using criteria then we need to **create the ProjectionList with the multiple properties** and then we need to add that Projectionist to the criteria.

```

public class ProjectionsMultipleColumns {
    public static void main(String[] args) {
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");
        SessionFactory factory = cfg.buildSessionFactory();
        Session session = factory.openSession();

        Criteria criteria = session.createCriteria(BankBo.class);
        Criterion cn = Restrictions.gt("balance", new Double(2000));
        criteria.add(cn);

        ProjectionList projectionList = Projections.projectionList();
        projectionList.add(Projections.property("accname"));
        projectionList.add(Projections.property("balance"));
        criteria.setProjection(projectionList);

        List list = criteria.list();
        Iterator it = list.iterator();
        while (it.hasNext()) {
            Object[] o = (Object[]) it.next();
            System.out.println(o[0] + " : " + o[1]);
        }
        session.close();
        factory.close();
    }
}

```

```

Hibernate: select this_.accname as y0_, this_.balance as y1_ from bank this_ where
this_.balance>?
Surya : 3000.0
Rakesh : 4000.0
CHANDU : 5000.0

```

## Difference between HQL and CQL

HQL (Hibernate Query Language)	CQL (Criteria Query Language)
We can perform <b>both select and non-select operations</b>	Criteria <b>is only for selecting</b> the data, we cannot perform non-select operations
suitable for executing <b>Static Queries</b>	suitable for executing <b>Dynamic Queries</b>
Takes less time to execute	Takes more time to execute

## 9. Native SQL Queries

HQL or Criteria queries able to execute almost any SQL query you want. However, many developers are complaint about the Hibernate's generated SQL statement is slow and more prefer to generated their own SQL (native SQL) statement.

Hibernate provide a **createSQLQuery()** method to let you call your native SQL statement directly. Your application will create a native SQL query from the session with the **createSQLQuery()** method on the Session interface.:.

```
public SQLQuery createSQLQuery(String sqlString) throws HibernateException
```

- By using Native SQL, we can perform both **SELECT & NON- SELECT** operations on the data
- We can use the database specific keywords (commands), to get the data from the database
- The main drawback of Native SQL is it makes the hibernate application as **database dependent**.

### Working with Native SQL Queries

Even though we are selecting complete objects from the database **we need to type cast into Object[]** array only, **not into our pojo class type**, because we are giving direct table, column names in the Native SQL Query so it doesn't know our class name

```
SQLQuery qry = session.createSQLQuery("select * from EMPLOYEE");
// Here EMPLOYEE is the table in the database...
List l = qry.list();
Iterator it = l.iterator();
while(it.hasNext())
{
Object row[] = (Object[])it.next();
-----
}
```

In the above code, we typecast into the **Object[]** , if we want to type cast into our POJO class , then we need to go with **entityQuery** concept. to make the query as an entityQuery, we need to call **addEntity()** method

```
//We are letting hibernate to know our pojo class too
SQLQuery q=session.createSQLQuery("select *from
EMPLOYEE").addEntity(EmployeeBo.class);
List l = q.list();
Iterator it = l.iterator();
while(it.hasNext())
{
    EmployeeBo s = (EmployeeBo)it.next();
-----
}
```

## Example

```
public class NativeSqlDemo {
    public static void main(String[] args) {
        // 1.Load Configuration
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");
        // 2.Create Session
        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        System.out.println("1.Simple Native SQL\n ======");
        SQLQuery query1 = session.createSQLQuery("select *from EMPLOYEE");
        List list1 = query1.list();
        Iterator it1 = list1.iterator();
        while (it1.hasNext()) {
            Object[] ob = (Object[]) it1.next();
            System.out.println(ob[0] + ", " + ob[1] + ", " + ob[2]);
        }
        System.out.println("2.Native SQL with entityQuery\n =====");
        SQLQuery query2 = session.createSQLQuery("select *from EMPLOYEE");
        query2.addEntity(EmployeeBo.class);
        List list2 = query2.list();
        Iterator it2 = list2.iterator();
        while (it2.hasNext()) {
            EmployeeBo bo = (EmployeeBo) it2.next();
        }
        session.close();
        sf.close();
    }
}
```

```
1.Simple Native SQL
=====
1, SATYA, VIJAYAWADA
2, SURYA, HYDERABAD
2.Native SQL with entityQuery
=====
1, SATYA, VIJAYAWADA
2, SURYA, HYDERABAD
```

## 10. Named Queries

if we want to execute the **same queries for multiple times** in our program then we can use the **Named Queries mechanism**

- In this Named Queries concept, **we use some name for the query configuration, and that name will be used whenever the same query is required to execute**
- If you want to create Named Query, in hibernate mapping file we need to configure a **query** by putting some name for it.
- In HQL, we need to use **<query name="query\_name">** to configure query

```
<query name="bankHQLQuery">
<! [CDATA[from BankBo b where b.balance>:bal ]]>
</query>
```

- In Native SQL, we need to use **<sql-query name="query\_name">** to configure query

```
<sql-query name="bankNativeQuery">
    select * from Employee
</sql-query>
```

- In our main program, we need to use **getNamedQuery()** given by **session** interface, for getting the **Query** reference and we need to execute that query by calling **list()**

```
Query qry = session.getNamedQuery("Name given in hib-mapping-xml");
qry.setParameter("bal",new Integer(3000));
List l = qry.list();
```

### Example 1 : HQL Named Query Example

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration SYSTEM
"hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/smlcodes</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>
<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">update</property>
        <mapping resource="EmployeeBo.hbm.xml" />
        <mapping resource="BankBo.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

### BankBo.java

```
public class BankBo {
    private int accno;
    private String accname;
    private double balance;
    //Setters & getters
```

### BankBo.hbm.xml

```
<hibernate-mapping>
    <class name="hql.BankBo" table="bank">
        <id name="accno" column="accno" />
        <property name="accname" column="accname" />
        <property name="balance" column="balance" />
    </class>

    <query name="bankHQLQuery">
        <![CDATA[from hql.BankBo b where b.balance>:bal ]]>
    </query>
</hibernate-mapping>
```

### HQLNamedQuery.java

```
package namedQuery;
public class HQLNamedQuery {
    public static void main(String[] args) {
        // 1.Load Configuration
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");
        // 2.Create Session
        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        System.out.println("HQL-NamedQuery Example\n-----");
        Query qry = session.getNamedQuery("bankHQLQuery");
        qry.setParameter("bal", new Double(3000));
        List list = qry.list();
        Iterator it = list.iterator();
        while (it.hasNext()) {
            BankBo bo = (BankBo) it.next();
            System.out.println(bo.getAccname() + ", " + bo.getBalance());
        }
        session.close();
        sf.close();
    }
}
```

#### HQL-NamedQuery Example

-----

Hibernate: select bankbo0\_.accno as accno1\_, bankbo0\_.accname as accname1\_,  
 bankbo0\_.balance as balance1\_ from bank bankbo0\_ where bankbo0\_.balance>?  
 Rakesh, 4000.0  
 CHANDU, 5000.0

## Example 2 : Native SQL Named Query Example

### EmployeeBo.java

```
package bo;

public class EmployeeBo {
    private int eid;
    private String name;
    private String address;
    //Setters & getters
}
```

### EmployeeBo.hbm.xml

```
<hibernate-mapping>
    <class name="bo.EmployeeBo" table="employee">
        <id name="eid" column="eid">
            <generator class="uuid" />
        </id>
        <property name="name" column="name" />
        <property name="address" column="address" />
    </class>

    <sql-query name="employeeNativeQuery">
        select * from Employee
    </sql-query>
</hibernate-mapping>
```

### NamedQueryDemo.java

```
package namedQuery;
public class NamedQueryDemo {
    public static void main(String[] args) {
        // 1.Load Configuration
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");
        // 2.Create Session
        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();
        Query qry = session.getNamedQuery("employeeNativeQuery");
        List list = qry.list();
        Iterator it = list.iterator();
        while (it.hasNext()) {
            Object o[] = (Object[]) it.next();
            System.out.println(o[0] + ", " + o[1] + ", " + o[2]);
        }
        session.close();
        sf.close();
    }
}
```

HQL-NamedQuery Example-----  
Hibernate: select bankbo0\_.accno as accno1\_, bankbo0\_.accname as accname1\_, bankbo0\_.balance as balance1\_ from bank bankbo0\_ where bankbo0\_.balance>?  
Rakesh, 4000.0  
CHANDU, 5000.0

# 11. Hibernate Relationships

In JAVA we have **three types** of relationships

1. **IS – A**
2. **HAS –A**
3. **Uses – A**

**1.Is-a** relationship is one in which **data members of one class is obtained into another class through the concept of inheritance.**

```
class B{  
}  
  
public class A extends B{  
}
```

**2.Has-a** relationship is one in which an **object of one class is created as a data member in another class.**

```
class Emp{  
    int a =100;  
}  
  
public class Demo {  
    private Emp emp;   
    public static void main(String[] args) {  
        |  
    }  
}
```

**3.Uses-a** relationship is one in which an **Object of one class is created inside a method of another class.**

```
class Emp {  
    int a = 100;  
}  
  
public class Demo {  
    public void get() {  
        Emp ob = new Emp();   
    }  
}
```

**NOTE: But in Database We don't have is-a,has-a,uses-a relationships like in java**

## Relationships in Hibernate

In database we don't have above mentioned relationships, but we can form the relationship between database tables by using **Primary Key (P.K), Foreign Key (F.K)**. **We can only form HAS-A relationship in hibernate.**

We have four types of relationships

1. One-To-One                    (**P.K → P.K, F.K(Unique) → P.K**)
2. One-To-Many                 (**P.K → F.K**)
3. Many-To-One                 (**F.K → P.K**)
4. Many-To-Many                (**Table1 → LinkTable → Table2**)

If you see above relations One→P.K, Many→F.K. Just kidding ☺

## Mapping Directions

Each relationship can be achieved in two directions

**1.Unidirectional** relationship refers to the case when relationship between two objects can be accessed by **one way only**. i.e. from Library we can access books, from book we can't access Library, not vice versa.

```
class Book{  
    Long bookId;  
    String bookName;  
    .....  
}  
class Library{  
    Long libraryId;  
    String libraryName;  
    Set<Book> bookSet;  
}
```

**2.Bidirectional** relationship refers to the case when relationship between two objects can be accessed both ways. i.e. From Library we can access books, from book we can access Library, vice versa.

```
class Book{  
    Long bookId;  
    String bookName;  
    Library library;  
    .....  
}  
class Library {  
    Long libraryId;  
    String libraryName;  
    Set<Book> bookSet;  
}
```

Relationship	Explanation	Example
<b>One-To-One</b>	Each record in one table is related to exactly one record in the second table and vice versa. The other side could also be a zero record.	A car has only one engine.
<b>One-To-Many (or) Many-To-One</b>	Each record in one table is related to zero or more records in the second table.	A movie has many actors (one-to-many); an actor can act in many movies (many-to-one).
<b>Many-To-Many</b>	Each record in either of the tables is related to zero or more records in the other table	Each student can enroll in multiple courses, and each course can have many students registered.

**REMEMEBER: Object means one row in hibernate terminology**

## 1. One-to-One Relationship Mapping Example

Using our **Car and Engine example**, we develop a one-to-one association. There are two ways of establishing a one-to-one association, **using a primary key using a foreign key**.

### Example: OneToOne Mapping (BiDirectional)

```
CREATE TABLE CAR (
CAR_ID int(10) NOT NULL,
NAME varchar(20) DEFAULT NULL,
COLOR varchar(20) DEFAULT NULL,
PRIMARY KEY (CAR_ID));

CREATE TABLE ENGINE (
CAR_ID int(10) NOT NULL,
SIZE varchar(20) DEFAULT NULL,
MODEL varchar(20) DEFAULT NULL,
PRIMARY KEY (CAR_ID),
FOREIGN KEY (CAR_ID) REFERENCES car (CAR_ID));
```

- **CAR** table with a **CAR\_ID** as the primary key.
- **ENGINE** table, primary key is a **CAR\_ID**.
- **ENGINE** table has a foreign key constraint pointing to the primary key of the CAR table. So, an engine will always be created with the same id as that of a car. Thus, we say both tables share the same primary key.

### Car.java

```
package onetoone;
public class Car {
    private int id;
    private String name;
    private String color;
    private Engine engine;
    //Setters & Getters
}
```

### Engine.java

```
package onetoone;
public class Engine {
    private int id = 0;
    private String model = null;
    private String size = null;
    // this engine is fitted to a car
    private Car car = null;
    //Setters & Getters
}
```

### Car.hbm.xml

```
<hibernate-mapping package="onetoone">
    <class name="Car" table="CAR">
        <id name="id" column="CAR_ID">
            <generator class="assigned" />
        </id>
        <property name="name" column="NAME" />
        <property name="color" column="COLOR" />
        <one-to-one name="engine" class="Engine" cascade="all" />
    </class>
</hibernate-mapping>
```

### Engine.hbm.xml

```
<hibernate-mapping package="onetoone">
    <class name="Engine" table="ENGINE">
        <id name="id" column="CAR_ID">
            <generator class="foreign">
                <param name="property">car</param>
            </generator>
        </id>
        <one-to-one name="car" class="Car" constrained="true" />
        <property name="size" column="SIZE" />
        <property name="model" column="model" />
    </class>
</hibernate-mapping>
```

### hibernate.cfg.xml

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

### OneToOneTest.java

```
package onetoone;
public class OneToOneTest {
public static void main(String[] args) {
    Configuration cfg = new Configuration();
    cfg.configure("hibernate.cfg.xml");
    SessionFactory sf = cfg.buildSessionFactory();
    Session session = sf.openSession();

    Car car = new Car();
    // Remember, we are using application generator for ids
    car.setId(1);
    car.setName("SWIFT");
    car.setColor("BLUE");
    // Next, create an instance of engine and set values.
    // Note: you are not setting the id!
    Engine engine = new Engine();
    engine.setModel("2009");
    engine.setSize("85KG");
    // Now we associate them together using the setter on the car
    car.setEngine(engine);
    engine.setCar(car);
    // Lastly, we are persisting them

    Transaction tx = session.beginTransaction();
    session.save(car);
    session.save(engine);
    tx.commit();
    System.out.println("Success");
}
}
```

```
mysql> select * from car;
+-----+-----+-----+
| CAR_ID | NAME          | COLOR |
+-----+-----+-----+
|      1 | SWIFT         | BLUE  |
+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from engine;
+-----+-----+-----+
| CAR_ID | size        | MODEL |
+-----+-----+-----+
|      1 | 85KG        | 2009  |
+-----+-----+-----+
```

## 2.One-to-Many / Many-to-One Relationship

To achieve one-to-many between two pojo classes in the hibernate, then the following two changes are required

- In the parent pojo class, we need to take a **collection property**, the collection can be either **Set**, **List**, **Map**.

```
public class Movie {  
    private int mid;  
    private String title;  
    private Set<Actor> actors;  
    //Setters & Getters  
}
```

- In the mapping file of that parent pojo class, we need to configure the collection

```
<set name="actors" table="actor" cascade="all">  
    <key column="mid" not-null="true" />  
    <one-to-many class="Actor" />  
</set>
```

In this example we are taking Movie & Actor table, the relation between them is "**one movie consists of one or more (i.e., many) actors**"

```
CREATE TABLE `movie` (  
`mid` INT(10) NOT NULL AUTO_INCREMENT,  
`title` VARCHAR(10) NULL DEFAULT NULL,  
PRIMARY KEY (`mid`)  
)  
  
CREATE TABLE `actor` (  
`actorid` INT(10) NOT NULL AUTO_INCREMENT,  
`name` VARCHAR(20) NULL DEFAULT NULL,  
`age` INT(10) NULL DEFAULT NULL,  
`mid` INT(10) NULL DEFAULT NULL,  
PRIMARY KEY (`actorid`),  
INDEX `FK585A9F550821B2A` (`mid`),  
INDEX `FK585A9F578674FF6` (`mid`),  
CONSTRAINT `FK585A9F550821B2A` FOREIGN KEY (`mid`) REFERENCES `movie` (`mid`)  
)
```

```
mysql> select * from actor  
-> ;  
+----+-----+-----+-----+  
| mid | actorid | name   | age   |  
+----+-----+-----+-----+  
| 501 |      101 | AMIR KHAN | 42   |  
| 501 |      102 | R. MADHAVAN | 36   |  
| 501 |      103 | KAREENA KAPOOR | 31   |  
+----+-----+-----+-----+
```

### Actor.java

```
package onetomany;
public class Actor {
    private int actorid;
    private String actorname;
    private int age;
    //Setters & getters
}
```

### Movie.java

```
package onetomany;
public class Movie {
    private int mid;
    private String title;
    private Set<Actor> actors;
    //Setters & Getters
}
```

### Actor.hbm.xml

```
<hibernate-mapping package="onetomany">
    <class name="Actor" table="actor">
        <id name="actorid" column="actorid">
            <generator class="assigned" />
        </id>
        <property name="actorname" column="name" />
        <property name="age" column="age" />
    </class>
</hibernate-mapping>
```

### Movie.hbm.xml

```
<hibernate-mapping package="onetomany">
    <class name="Movie" table="movie">
        <id name="mid" column="mid">
            <generator class="assigned"/>
        </id>

        <property name="title" column="title" />
        <set name="actors" table="actor" cascade="all">
            <key column="mid" not-null="true" />
            <one-to-many class="Actor" />
        </set>
    </class>
</hibernate-mapping>
```

- In order to transfer operations on parent object to child object we need to add **cascade** attribute
- By default, cascade value is none, it means even though relationship is exist, the operations we are doing on parent will not transfer to child.
- In above xml, we used **cascade = " all"** means all operations at parent object will be transfer to child

- In the mapping file, we need to use **<key /> element to configure foreign key column name**, in this example “*mid*” is foreign key
- **<one-to-many>** is child class with which relation been done, in our example **Actor** is the child class

### OneToManyTest.java

```

package onetomany;

import java.util.HashSet;
import java.util.Set;

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

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

    Configuration cfg = new Configuration();
    cfg.configure("hibernate.cfg.xml");
    SessionFactory sf = cfg.buildSessionFactory();
    Session session = sf.openSession();

    Actor amir = new Actor();
    amir.setActorname("AMIR KHAN");
    amir.setAge(42);
    amir.setActorid(101);

    Actor madhav = new Actor();
    madhav.setActorname("R. MADHAVAN");
    madhav.setAge(36);
    madhav.setActorid(102);

    Actor kareena = new Actor();
    kareena.setActorname("KAREENA KAPOOR");
    kareena.setAge(31);
    kareena.setActorid(103);

    Set<Actor> actors =new HashSet<Actor>();
    actors.add(amir);
    actors.add(madhav);
    actors.add(kareena);

    Movie movie = new Movie();
    movie.setTitle("3 IDIOTS");
    movie.setActors(actors);
    movie.setMid(501);

    Transaction tx = session.beginTransaction();
    session.save(movie);
    tx.commit();
    System.out.println("Success");
}
}

```

### 3.Many to Many Relationship

Applying many to many relationships between two pojo class objects is nothing but applying **one to many relationships on both sides**, which tends to Bi-Directional i mean many to many.

- when ever we are applying many to many relationships between two pojo class objects, **on both sides we need a collection property**
- While applying **many to many** relationships between pojo classes, a **mediator table is mandatory in the database**, to store primary key as foreign key both sides, we call this table as Join table

In the example of **Student and Course**, the relationship can be many-to-many: a student can take many courses, while a course may consist of many students

```
CREATE TABLE `course` (
`courseId` INT(11) NOT NULL AUTO_INCREMENT,
`courseName` VARCHAR(50) NULL DEFAULT NULL,
PRIMARY KEY (`courseId`)
)

CREATE TABLE `student` (
`studentId` INT(11) NOT NULL AUTO_INCREMENT,
`studentName` VARCHAR(50) NULL DEFAULT NULL,
PRIMARY KEY (`studentId`)
)

// mediator table
CREATE TABLE `student_course` (
`COURSE_ID` INT(10) NOT NULL,
`STUDENT_ID` INT(10) NOT NULL,
PRIMARY KEY (`COURSE_ID`, `STUDENT_ID`),
INDEX `FKCB6FBEBFAD895D0F` (`COURSE_ID`),
INDEX `FKCB6FBEBF88820545` (`STUDENT_ID`),
CONSTRAINT `fk_c_id` FOREIGN KEY (`COURSE_ID`) REFERENCES `course` (`courseId`),
CONSTRAINT `fk_s_id` FOREIGN KEY (`STUDENT_ID`) REFERENCES `student` (`studentId`)
)
```

Course.java

```
package manytomany;

public class Course {
    private int courseId;
    private String courseName;
    private Set<Student> students;
    //Setters & Getters
}
```

### Student.java

```
package manytomany;

public class Student {
    private int studentId;
    private String studentName;
    private Set<Course> courses;
//Setters & Getters
}
```

### Course.hbm.xml

```
<hibernate-mapping package="manytomany">
    <class name="Course" table="course">
        <id name="courseId" column="courseId">
            <generator class="assigned" />
        </id>
        <property name="courseName" column="courseName" />
        <set name="students" table="STUDENT_COURSE" inverse="true"
            cascade="all">
            <key column="COURSE_ID" />
            <many-to-many column="STUDENT_ID" class="Student" />
        </set>
    </class>
</hibernate-mapping>
```

### Student.hbm.xml

```
<hibernate-mapping package="manytomany">
    <class name="Student" table="student">
        <id name="studentId" column="studentId">
            <generator class="assigned" />
        </id>
        <property name="studentName" column="studentName" />
        <set name="courses" table="STUDENT_COURSE" cascade="all">
            <key column="STUDENT_ID" />
            <many-to-many column="COURSE_ID" class="Course" />
        </set>
    </class>
</hibernate-mapping>
```

```
package manytomany;
public class ManyToManyTest {
public static void main(String[] args) {

    Configuration cfg = new Configuration();
    cfg.configure("hibernate.cfg.xml");
    SessionFactory sf = cfg.buildSessionFactory();
    Session session = sf.openSession();

    Student s1 = new Student();
    s1.setStudentId(101);
    s1.setStudentName("SATYA");
```

```

Student s2 = new Student();
s2.setStudentId(102);
s2.setStudentName("ARJUN");

Course c1 = new Course();
c1.setCourseId(301);
c1.setCourseName("JAVA");

Course c2 = new Course();
c2.setCourseId(302);
c2.setCourseName(".NET");

Set<Student> students = new HashSet<Student>();
students.add(s1);
students.add(s2);
c1.setStudents(students);
c2.setStudents(students);

Set<Course> courses = new HashSet<Course>();
courses.add(c1);
courses.add(c2);
s1.setCourses(courses);
s2.setCourses(courses);

Transaction tx = session.beginTransaction();
session.save(s1);
session.save(s2);
tx.commit();
System.out.println("Success");
}
}
}

mysql> select * from student;
+-----+-----+
| studentId | studentName |
+-----+-----+
| 101 | SATYA |
| 102 | ARJUN |
+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from course;
+-----+-----+
| courseId | courseName |
+-----+-----+
| 301 | JAVA |
| 302 | .NET |
+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from student_course;
+-----+-----+
| COURSE_ID | STUDENT_ID |
+-----+-----+
| 301 | 101 |
| 301 | 102 |
| 302 | 101 |
| 302 | 102 |
+-----+-----+

```

## Cascade Attribute In Hibernate

Cascade attribute is mandatory, whenever we apply relationship between objects, cascade attribute transfers operations done on one object onto its related child objects. default value of **cascade = "none"** means no operations will be transferred to the child class

If we write **cascade = "all"** then changes like insert, delete, update at parent object will be effected to child object.

Cascade having following values

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

## 12. Hibernate Cache

Every fresh session having its own **cache memory**, **Caching is a mechanism for storing the loaded objects into a cache memory**. The advantage of cache mechanism is, whenever again we want to load the same object from the database then instead of hitting the database once again, it loads from the local cache memory only, **so that the no. of round trips between an application and a database server got decreased**. It means caching mechanism increases the performance of the application.

In hibernate we have two levels of caching

1. **First Level Cache** (Session Cache)
2. **Second Level Cache** (Session Factory Cache/ JVM Level Cache)

### 1. First Level Cache

- By default, for each hibernate application, **the first level cache is automatically enabled. We can't Enable/Disable first level cache**
- the first level cache is associated with the **session** object and **scope** of the cache **is limited to one session only**
- When we load an object for the first time from the database then the object will be loaded from the database and the loaded object will be stored in the cache memory maintained by that session object
- If we load the same object once again, with in the same session, then the object will be loaded from the local cache memory not from the database
- If we load the same object by opening other session, then again the object will load from the database and the loaded object will be stored in the cache memory maintained by this new session

#### Example:

```
Session session = factory.openSession();
Object ob1 = session.get(Actor.class, new Integer(101)); //1

Object ob2 = session.get(Actor.class, new Integer(101));//2
Object ob3 = session.get(Actor.class, new Integer(101));//3
session.close();//4

Session ses2 = factory.openSession();
Object ob5 = ses2.get(Actor.class, new Integer(101));//5
```

1, We are loaded object with id 101, now it will load the object from the database only as its the first time, and keeps this object in the session cache

2,3 i tried to load the same object 2 times, but here the object will be loaded from the stored cache only not from the database, as we are in the same session

4, we close the first session, so the cache memory related this session also will be destroyed

5, again i created one new session and loaded the same object with id 101, but this time hibernate will loads the object from the database

if we want to remove the objects that are stored in the cache memory, then we need to call either **evict()** or **clear()** methods

## **2. Second Level Cache**

Whenever we are loading any object from the database, then hibernate verify whether that object is available in the local cache(**first level cache**) memory of that particular session, if not available then hibernate verify whether the object is available in global cache(**second level cache**), if not available then hibernate will hit the database and loads the object from there, and then **first stores in the local cache** of the session , then in the global cache

**SessionFactory** holds the second level cache data. It is global for all the session objects and not enabled by default.

Different vendors have provided the implementation of Second Level Cache

- 1. EH Cache**
- 2. OS Cache**
- 3. Swarm Cache**
- 4. JBoss Cache**

To enable second level cache in the hibernate, then the following **3** changes are required

1. **Add provider class** in hibernate configuration file

```
<property name="hibernate.cache.provider_class">
    org.hibernate.cache.EhCacheProvider
</property>
```

2. **Configure cache element** for a class in hibernate mapping file

```
<cache usage="read-only" />
```

- o **read-only**: caching will work for read only operation.
- o **nonstrict-read-write**: caching will work for read and write but one at a time.
- o **read-write**: caching will work for read and write, can be used simultaneously.
- o **transactional**: caching will work for transaction.

3. create xml file called **ehcache.xml** and place where you have mapping and configuration xml's

**Example:**

```
public class Employee {  
    private int eid;  
    private String name;  
    private String address;  
    //Setters & Getters  
}
```

**Employee.hbm.xml**

```
<hibernate-mapping package="cache">  
    <class name="Employee" table="employee">  
        <cache usage="read-only" />  
        <id name="eid" column="eid">  
            <generator class="native"></generator>  
        </id>  
        <property name="name"></property>  
        <property name="address"></property>  
    </class>  
</hibernate-mapping>
```

**ehcache.xml**

```
<?xml version="1.0"?>  
<ehcache>  
    <defaultCache maxElementsInMemory="100" eternal="false"  
                  timeToIdleSeconds="120" timeToLiveSeconds="200" />  
    <cache name="cache.Employee" maxElementsInMemory="100"  
          eternal="false" timeToIdleSeconds="5" timeToLiveSeconds="200" />  
</ehcache>
```

**hibernate.cfg.xml**

```
<hibernate-configuration>  
    <session-factory>  
        <property> Driver Class, URL, Username, password, etc </property>  
        <property  
name="cache.provider_class">org.hibernate.cache.EhCacheProvider</property>  
        <property name="hibernate.cache.use_second_level_cache">true</property>  
        <mapping resource="cache/employee.hbm.xml" />  
    </session-factory>  
</hibernate-configuration>
```

### CacheDemo.java

```
package cache;

import org.hibernate.*;
import org.hibernate.cfg.*;
public class CacheDemo {

    public static void main(String[] args) {

        //1.Load Configuration
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

        //2.Create Session
        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();

        //3.Perform Operations
        Object ob = session.load(Employee.class, new Integer(1));
        Employee bo = (Employee) ob;

        System.out.println("SELECTED DATA\n =====");
        System.out.println("SNO : "+bo.getEid());
        System.out.println("NAME : "+bo.getName());
        System.out.println("ADDRESS : "+bo.getAddress());
    }
}
```

```
mysql> select * from employee;
+---+-----+-----+
| eid | name | address |
+---+-----+-----+
|   1 | SATYA | VIJAYAWADA |
|   2 | SURYA | HYDERABAD |
|   3 | RAVI  | PUNE   |
+---+-----+-----+
3 rows in set (0.00 sec)
```

# 13. Hibernate with Annotations

---

The EJB 3 standard annotations are contained in the **javax.persistence** package, so we import this package.

- Use annotations in POJO classes. These classes are called Entity Bean Classes
- No need of xml files
- We use **AnnotationConfiguartion** class instead of Configuration class

```
Configuration cfg = new AnnotationConfiguartion();
```

- From Hibernate 4.x version Configuration is enough for both annotation and xml configuration
- We have to configure POJO class in hbm.xml

```
<mapping class="bean.Student">
```

## Commonly used Annotations in Hibernate

### 1. @Entity

- Annotation marks this class as an entity.
- We have to place this annotation at the top of class

### 2. @Table

- Specifies Table to be connect with this class. If you don't use **@Table** annotation, hibernate will use the **class name as the table name by default**.
- We have to **place this annotation at the top of class**.

### 3. @Id

Every table has a primary key; we can make the data member as Primary Key using @Id annotation.

### 4. @GeneratedValue

It will generate the Primary Key/ ID values automatically

### 5. @Column

- This Annotation specifies the details of the column for this property or field.
- If @Column is not specified, property name will be used as the column name by default.

### 6. @Transient

We can declare the data members which are **not have columns in database table**

## 7.@ManyToMany

- **Cascade:** Marks this field as the owning side of the many-to-many relationship and cascade modifier specifies which operations should cascade to the inverse side of relationship
- **mappedBy:** This modifier holds the field which specifies the inverse side of the relationship

## 8.@JoinTable

- **Name:** For holding this many-to-many relationship, maps this field with an intermediary database join table specified by name modifier
- **joinColumns:** Identifies the owning side of columns which are necessary to identify a unique owning object
- **inverseJoinColumns:** Identifies the inverse (target) side of columns which are necessary to identify a unique target object

## 9.@JoinColumn

Maps a join column specified by the name identifier to the relationship table specified by @JoinTable

### Example: CURD operations using Annotations

```
CREATE TABLE `studenttable` (
`sno` INT(11) NOT NULL AUTO_INCREMENT,
`name` VARCHAR(50) NULL DEFAULT NULL,
`address` VARCHAR(50) NULL DEFAULT NULL,
PRIMARY KEY (`sno`)
)
```

### StudentBo.java

```
package annotations;

@Entity
@Table(name="studenttable")
public class StudentBo {

@Id
@Column(name="sno")
@GeneratedValue(strategy=GenerationType.AUTO)
private int sno; //PRIMARY_KEY

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

@Column           //By default it will take datamember name
private String address;
```

```

@Transient
private String iamnotindatabase;
//This column not their in db

public int getSno() {
    return sno;
}

public void setSno(int sno) {
    this.sno = sno;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getAddress() {
    return address;
}

public void setAddress(String address) {
    this.address = address;
}

public StudentBo( String name, String address) {
    super();

    this.name = name;
    this.address = address;
}
public StudentBo() {
    super();
}
}

```

```

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/smlcodes</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>

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

        <property name="cache.provider_class">org.hibernate.cache.EhCacheProvider</property>
        <property name="hibernate.cache.use_second_level_cache">true</property>

        <mapping class="annotations.StudentBo"/>
    </session-factory>
</hibernate-configuration>

```

### AnnotationExample.java

```
package annotations;
public class AnnotationExample {
    public static void main(String[] args) {

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

        SessionFactory sf = cfg.buildSessionFactory();
        Session session = sf.openSession();
        Transaction tx = session.beginTransaction();
        System.out.println("1.Save Operation");
        System.out.println("=====");
        StudentBo e1 = new StudentBo("SATYA", "HYD");
        StudentBo e2 = new StudentBo("RAM", "BANGLORE");
        StudentBo e3 = new StudentBo("KIRAN", "MUMBAI");
        session.save(e1);
        session.save(e2);
        session.save(e3);

        System.out.println("2.Select Operation");
        System.out.println("=====");
        List<StudentBo> ob = session.createQuery("FROM StudentBo").list();

        for (StudentBo e : ob) {
            System.out.println(e.getSno() + ", " + e.getName() + ", " + e.getAddress());
        }
        tx.commit();
        session.close();
        sf.close();
    }
}
```

```
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.
1.Save Operation
=====
Hibernate: insert into studenttable (name, address) values (?, ?)
Hibernate: insert into studenttable (name, address) values (?, ?)
Hibernate: insert into studenttable (name, address) values (?, ?)
2.Select Operation
=====
Hibernate: select studentbo0_.sno as sno0_, studentbo0_.name as name0_,
studentbo0_.address as address0_ from studenttable studentbo0_
1, SATYA, HYD
2, RAM, BANGLORE
3, KIRAN, MUMBAI
```

## Mappings using Annotations

### 1.one-to-one mapping using Annotations

Car.java

```
package annotations.onetoone;

@Entity
@Table(name="car")
public class Car {

    @Id
    @Column(name="CAR_ID")
    private int id;

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

    @Column(name="COLOR")
    private String color;

    //Setters & Getters
}
```

Engine.java

```
package annotations.onetoone;

@Entity
@Table(name="engine")
public class Engine {

    @Id
    private int id = 0;

    @Column
    private String model = null;

    @Column
    private String size = null;

    @OneToOne(targetEntity=Car.class,cascade=CCascadeType.ALL)
    @JoinColumn(name="CAR_ID",referencedColumnName="CAR_ID")
    private Car car = null;

    //Setters & Getters
}
```

## OneToOneTest.java

```
package annotations.onetoone;

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

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

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

        Car car = new Car();
        car.setId(2);
        car.setName("BENZ");
        car.setColor("RED");

        Engine engine = new Engine();
        engine.setModel("2209");
        engine.setSize("815KG");
        engine.setCar(car);

        Transaction tx = session.beginTransaction();
        session.save(car);
        session.save(engine);
        tx.commit();
        System.out.println("Success");
    }
}
```

```
mysql> select * from car;
+-----+-----+-----+
| CAR_ID | NAME | COLOR |
+-----+-----+-----+
|      2 | BENZ | RED   |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from engine;
+-----+-----+-----+-----+
| CAR_ID | size  | MODEL | id   |
+-----+-----+-----+-----+
|      2 | 815KG | 2209 |    0 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

## 2.one-to-many Mapping Using Annotations

Actor.java

```
package annotations.onetomany;

@Entity
@Table(name="actor")
public class Actor {
    @Id
    @Column
    private int actorid;

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

    @Column
    private int age;

    //Setters & Getters

}
```

Movie.java

```
package annotations.onetomany;

@Entity
@Table(name="movie")
public class Movie {

    @Id
    @Column
    private int mid;

    @Column
    private String title;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name="mid", referencedColumnName="mid")
    private Set<Actor> actors;
    //Setters & Getters
}
```

## OneToManyTest.java

```
package annotations.onetomany;

public class OneToManyTest {
public static void main(String[] args) {
    Configuration cfg = new AnnotationConfiguration();
    cfg.configure("hibernate.cfg.xml");

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

    Actor amir = new Actor();
    amir.setActorname("PRABAS");
    amir.setAge(36);
    amir.setActorid(106);

    Actor madhav = new Actor();
    madhav.setActorname("DAGGUBATI RANA");
    madhav.setAge(31);
    madhav.setActorid(107);

    Actor kareena = new Actor();
    kareena.setActorname("KATTAPPA");
    kareena.setAge(61);
    kareena.setActorid(108);

    Set<Actor> actors =new HashSet<Actor>();
    actors.add(amir);
    actors.add(madhav);
    actors.add(kareena);

    Movie movie = new Movie();
    movie.setTitle("BAAHUBALI");
    movie.setActors(actors);
    movie.setMid(502);

    Transaction tx = session.beginTransaction();
    session.save(movie);
    tx.commit();
    System.out.println("Success");
}
}
```

```
mysql> select * from movie;
+---+-----+
| mid | title |
+---+-----+
| 502 | BAAHUBALI |
+---+-----+
1 row in set (0.00 sec)

mysql> select * from actor;
+---+-----+-----+-----+
| mid | actorid | name      | age   |
+---+-----+-----+-----+
| 502 |     106 | PRABAS   | 36    |
| 502 |     107 | DAGGUBATI RANA | 31    |
| 502 |     108 | KATTAPPA | 61    |
+---+-----+-----+-----+
3 rows in set (0.00 sec)
```

### 3.Many-to-many Mapping Using Annotations

Course.java

```
package annotations.manytomany;

@Entity
@Table(name="course")
public class Course {

    @Id
    @Column
    private int courseId;

    @Column
    private String courseName;

    @ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
    @JoinTable(name = "student_course", joinColumns = {
        @JoinColumn(name = "COURSE_ID", nullable = false, updatable = false) },
    inverseJoinColumns = { @JoinColumn(name = "STUDENT_ID",
        nullable = false, updatable = false) })
    private Set<Student> students;

    //Setters & Getters

}
```

Student.java

```
package annotations.manytomany;

@Entity
@Table(name="student")
public class Student {

    @Id
    @Column
    private int studentId;

    @Column
    private String studentName;

    @Column
    @ManyToMany(mappedBy = "students")
    private Set<Course> courses;

    //Setters & Getters
}
```

```

package annotations.manytomany;
public class ManyToManyTest {
public static void main(String[] args) {
    Configuration cfg = new AnnotationConfiguration();
    cfg.configure("hibernate.cfg.xml");
    SessionFactory sf = cfg.buildSessionFactory();
    Session session = sf.openSession();

    Student s1 = new Student();
    s1.setStudentId(105);
    s1.setStudentName("SACHIN");
    Student s2 = new Student();
    s2.setStudentId(112);
    s2.setStudentName("DHONI");

    Course c1 = new Course();
    c1.setCourseId(303);
    c1.setCourseName("DEVOPS");
    Course c2 = new Course();
    c2.setCourseId(304);
    c2.setCourseName("HACKING");

    Set<Student> students = new HashSet<Student>();
    students.add(s1);
    students.add(s2);
    c1.setStudents(students);
    c2.setStudents(students);

    Set<Course> courses = new HashSet<Course>();
    courses.add(c1);
    courses.add(c2);
    s1.setCourses(courses);
    s2.setCourses(courses);

    //we have to save the Course object, because we defiend M2M in Course class only
    Transaction tx = session.beginTransaction();
    session.save(c1);
    session.save(c2);
    tx.commit();
    System.out.println("Success");
}
}

```

mysql> select * from course;	mysql> select * from student_course;	mysql> select * from student;
+-----+-----+   courseId   courseName   +-----+-----+   303   DEVOPS     304   HACKING   +-----+-----+	+-----+-----+   COURSE_ID   STUDENT_ID   +-----+-----+   303   105     303   112     304   105     304   112   +-----+-----+	+-----+-----+   studentId   studentName   +-----+-----+   105   SACHIN     112   DHONI   +-----+-----+
2 rows in set <0.00 sec>		2 rows in set <0.00 sec>

## Errors & Solutions

---

```
org.hibernate.HibernateException: Could not parse configuration:  
hibernate.cfg.xml
```

It was failing because there was no internet connection / you are behind proxy. To solve this issues

- Extract hibernate3.jar file find hibernate-mapping-3.0.dtd, hibernate-configuration-3.0.dtd files
- Paste the above two files root folder of your project

In **hibernate.cfg.xml** change lines to

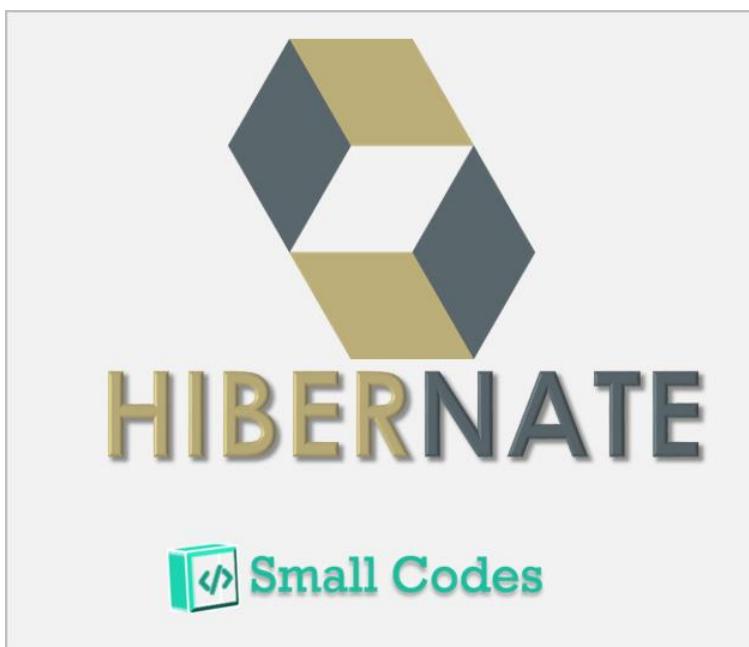
```
<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-configuration SYSTEM  
"hibernate-configuration-3.0.dtd">
```

In **<Class>.hbm.xml** change lines to

```
<?xml version="1.0"?>  
<!DOCTYPE hibernate-mapping SYSTEM  
"hibernate-mapping-3.0.dtd">
```

## References

---



<http://www.java4s.com/hibernate>

Mappings: O'Reilly Just.Hibernate.Jun.2014.ISBN.1449334377.pdf

# Hibernate Interview Questions

---

## 1.2 What are advantages of Hibernate?

- Lazy Loading
- Caching
- You do not need to maintain JDBC code , Hibernate takes care of it.
- You need to write less code
- It provides high level object oriented API

## 2 What is caching?

Anything you can do to minimize traffic between a database and an application server is probably a good thing. In theory, an application ought to be able to maintain a cache containing data already loaded from the database, and only hit the database when information has to be updated. When the database is hit, the changes may invalidate the cache

## 3 What are some core interfaces of hibernate?

- Configuration
- SessionFactory
- Session
- Transaction
- Query and Criteria interface

## 4 Difference between get() vs load() method in Hibernate? (detailed answer)

The key difference between get() and load() method is that

- **load()** will throw an exception if an object with id passed to them is not found
- **get()** will return null.

Another important difference is that load can return proxy without hitting the database unless required (when you access any attribute other than id) but get() always go to the database, so sometimes using load() can be faster than the get() method. It makes sense to use the load() method if you know the object exists but get() method if you are not sure about object's existence.

Parameter	get	load
Database retrieval	It always hits the database	It does not hit database
If null	If it does not get the object with id, it returns null	If it does get the object with id, it throws ObjectNotFoundException

Proxy	It returns real object	It returns proxy object
Use	If you are not sure if object with id exists or not, you can use get	If you are sure about existence of object, you can use load

## 5 What is the difference between save() and persist() method in Hibernate?

- **Serializable Object** **save()** returns a Serializable object
- **void persist()** method is void, so it doesn't return anything.

## 6 What is the difference between and merge and update?

Use update() if you are sure that the session does not contain an already persistent instance with the same identifier, and merge() if you want to merge your modifications at any time without consideration of the state of the session.

## 7 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.

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.hbm.xml -->
<hibernate-mapping>
    <class name="com.mkyong.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.mkyong.common.StockDailyRecord" />
        </set>
        ...
    </class>
</hibernate-mapping>
```

When you save or update the stock object

```
session.save(stock);
session.update(stock);
Copy
```

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.mkyong.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.mkyong.common.StockDailyRecord" />
        </set>
        ...
Copy
```

When you save or update the stock object

```
session.save(stock);
session.update(stock);
Copy
```

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.

## **8 Does SessionFactory is thread-safe in Hibernate? (detailed answer)**

**SessionFactory** is both **Immutable** and **thread-safe** and it has just one single instance in Hibernate application. It is used to create Session object and it also provide caching by storing SQL queries stored by multiple session. The second level cache is maintained at SessionFactory level.

## **9 Does Hibernate Session interface is thread-safe in Java? (detailed answer)**

No, Session object is not thread-safe in Hibernate and intended to be used with-in single thread in the application.

### **What is difference between getCurrentSession() and openSession() in Hibernate?**

**openSession()** When you call SessionFactory.openSession, it always create new Session object afresh and give it to you. As session objects are not thread safe, you need to create one session object per request in multithreaded environment and one session per request in web applications too.

**getCurrentSession()** When you call SessionFactory.getCurrentSession , it creates a new Session if not exists , else use same session which is in current hibernate context. It automatically flush and close session when transaction ends, so you do not need to do externally.If you are using hibernate in single threaded environment , you can use getCurrentSession, as it is faster in performance as compare to creating new session each time.

**You need to add following property to hibernate.cfg.xml to use getCurrentSession method**

```
<session-factory>
<!-- Put other elements here -->
<property name="hibernate.current_session_context_class"></property>
</session-factory>
```

If you do not configure above property, you will get error as below.

Exception in thread "main" org.hibernate.HibernateException: No CurrentSessionContext configured!

### **Can you declare Entity(Bean) class as final in hibernate?**

Yes, you can declare entity class as final but it is not considered as a good practice because hibernate uses proxy pattern for lazy initialization, If you declare it as final then hibernate won't be able to create sub class and won't be able to use proxy pattern, so it will limit performance and improvement options.

### **Does entity class (Bean) in hibernate require no arg constructor?**

Yes, Entity class in hibernate requires no arg constructor because Hibernate use reflection to create instance of entity class and it mandates no arg constructor in Entity class

## How do you log SQL queries issued by the Hibernate framework in Java application?

You can procedure the **show\_sql** property to log SQL queries delivered by the Hibernate framework

## What is named SQL query in Hibernate?

Named queries are SQL queries which are defined in mapping document using **<sql-query>** tag and called using **Session.getNamedQuery()** method.

```
<sql-query name="findStudentByRollNumber">
    <![CDATA[
        select * from Student student where student.rollNumber = :rollNumber
    ]]-->
</sql-query>
```

you can define named query in hibernate either by using annotations or XML mapping file, as I said above. **@NamedQuery** is used to define single named query and **@NameQueries** is used to define multiple named query in hibernate.

```
@NamedQueries({
    @NamedQuery(
        name = "findStockByStockCode",
        query = "from Stock s where s.stockCode = :stockCode"
    )
})
```

```
Query query = session.getNamedQuery("findStockByStockCode")
.setString("stockCode", "7277");
```

## Explain Criteria API

Criteria is a simplified API for retrieving entities by composing Criterion objects. This is a very convenient approach for functionality like “search” screens where there is a variable number of conditions to be placed upon the result set.

### Example:

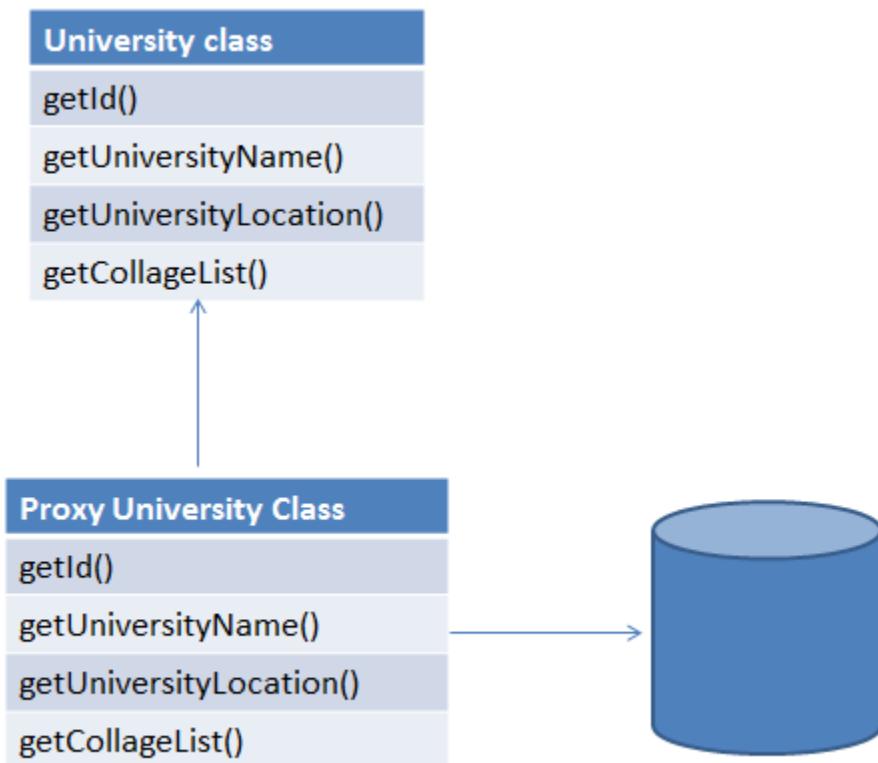
```
List employees = session.createCriteria(Employee.class)
.add(Restrictions.like("name", "a%"))
.add(Restrictions.like("address", "Boston"))
.addOrder(Order.asc("name"))
.list();
```

## How do you switch between relational databases without code changes?

Using Hibernate SQL Dialects, we can switch databases. Hibernate will generate appropriate hql queries based on the dialect defined.

## What is Hibernate proxy?

The proxy attribute enables lazy initialization of persistent instances of the class. Hibernate will initially return CGLIB proxies which implement the named interface. The actual persistent object will be loaded when a method of the proxy is invoked.



## What is automatic dirty checking?

Automatic dirty checking is a feature that saves us the effort of explicitly asking Hibernate to update the database when we modify the state of an object inside a transaction.

If Dirty-checking is enabled, if we forget to call save() before the commit, dirty-checking automatically saves the data into the database.

Consider the below code which loads a simple Entity from the database and updates it.

```
public static void testUpdate() {  
    Session session = sessionFactory.openSession();
```

```

        Transaction transaction = session.beginTransaction();
        Entity entity = (Entity) session.load(Entity.class, 1);
        entity.setData("Updating the data");
        transaction.commit();
        session.close();
    }

```

Although we haven't made any `session.update(entity)` call, the logs indicate that the database record was updated successful

### What is query cache in Hibernate?

Query cache can be used along with second level cache for improved performance. QueryCache actually stores the result of SQL query for future calls. Hibernate support various open source caching solution to implement Query cache e.g. EhCache

### What are two types of Collections in hibernate?

- Sorted Collection
- Ordered Collection

Parameter	Sorted Collection	Ordered Collection
Sorting	Sorted collection uses java's sorting API to sort the collection.	Ordered Collections uses order by clause while retrieval of objects
Default	It is enabled by default	It is not enabled by default, you need to enable it explicitly

### What is lazy loading in hibernate?

Sometimes you have two entities and there's a relationship between them. For example, you might have an entity called University and another entity called Student

```

public class University {
    private String id;
    private String name;
    private String address;
    private List<Student> students;
    // setters and getters
}

```

Now when you load a University from the database, JPA loads its id, name, and address fields for you. But you have two options for students: to load it together with the rest of the fields (i.e. eagerly) or to load it on-demand (i.e. lazily) when you call the university's `getStudents()` method.

```

@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    @JoinColumn(name="countryId")
    private List<Student> students;

```

**FetchType.LAZY:** It fetches the child entities lazily, that is, at the time of fetching parent entity it just fetches proxy (created by cglib or any other utility) of the child entities and when you access any property of child entity then it is actually fetched by hibernate.

**FetchType.EAGER:** it fetches the child entities along with parent.

Lazy initialization improves performance by avoiding unnecessary computation and reduce memory requirements.

Eager initialization takes more memory consumption and processing speed is slow.

`lazy="true/false in xml`

## Core Interfaces in Hibernate

*Configuration:*

*Application use configuration instance to specify the location of mapping docs and hibernate specific properties.*

*Application get instance of SessionFactory from Configuration.*

*SessionFactory :*

*Application get instance of session from SessionFactory .  
Not light weighted.  
Thread safe  
Single session factory for whole application*

*Session interface:*

*Session interface is primary interface use by hibernate applications.*

*Transaction:*

*Application get an instance of Transaction from Session.*

*Query and Criteria Interface:*

*The query interface allowed you to perform queries against the database and control how the query is executed.*

```

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

<hibernate-configuration>
  <session-factory>

    <property name="connection.driver_class">com.microsoft.jdbc.sqlserver.SQLServerDriver</property>
    <property name="connection.url">jdbc:microsoft:sqlserver://10.7.100.146:143;databaseName=YashTraining</property>
    <property name="connection.username">trainingyash</property>
    <property name="connection.password">trainingyash</property>

    <!-- JDBC connection pool (use the built-in) -->

    <property name="connection.pool_size">1</property>
    <property name="dialect">org.hibernate.dialect.SQLServerDialect</property>

    <!-- Echo all executed SQL to stdout -->

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

    <!--create-drop,create,update the database schema on startup -->

    <property name="hbm2ddl.auto">update</property>
    <mapping resource="HighScore.hbm.xml"/>
    <mapping resource="GameScore.hbm.xml"/>

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

```

### Hibernate Datasource Properties

#### Hibernate JDBC Properties

Property name	Purpose
hibernate.connection.driver_class	jdbc driver class
hibernate.connection.url	jdbc URL
hibernate.connection.username	database user
hibernate.connection.password	database user password
hibernate.connection.pool_size	maximum number of pooled connections

Property name	Purpose
hibernate.connection.datasource	datasource JNDI name
hibernate.jndi.url	URL of the JNDI provider (optional)
hibernate.jndi.class	class of the JNDI InitialContextFactory (optional)
hibernate.connection.username	database user (optional)
hibernate.connection.password	database user password (optional)

**Dialect:**

Type of Database Software using in our HIBERNATE application, with version

Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 11g	org.hibernate.dialect.Oracle10gDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
MySQL	org.hibernate.dialect.MySQLDialect
Microsoft SQL Server 2000	org.hibernate.dialect.SQLServerDialect
Microsoft SQL Server 2005	org.hibernate.dialect.SQLServer2005Dialect
Microsoft SQL Server 2008	org.hibernate.dialect.SQLServer2008Dialect
DB2	org.hibernate.dialect.DB2Dialect
HSQLDB	org.hibernate.dialect.HSQLDialect

org.hibernate.dialect.PostgreSQL
Progress
SAP DB
Sybase
Sybase Anywhere
DB2
HSQLDB
HypersonicSQL
Informix
Ingres
Interbase

**Mapping Configuration File**

1. You should save the mapping document in a file with the format <classname>.hbm.xml.
2. <hibernate-mapping> as the root element which contains all the <class> elements.
3. The <class> elements are used to define specific mappings from a Java classes to the database tables
4. The Java class name is specified using the "name" attribute
5. the database table name is specified using the "table" attribute.
6. The <id> element maps the unique ID attribute in class to the primary key of the database table.
7. The "name" attribute of the id element refers to the column in the database table.
8. The <property> element is used to map a Java class property to a column in the database table.
9. The "name" attribute of the element refers column in the database table.

```

<hibernate-mapping>
  <class name="persistance.IdName" table="idname">
    <id name = "id" column = "id"/>
    <property name = "name" column = "name"/>
  </class>
</hibernate-mapping>

```