**Disadvantages with JDBC edit test**

* Even though JDBC is DB independent but since it uses native SQL queries directly into it, it indirectly becomes DB dependent

Example:

MySQL & PostgreSQL: **SELECT** \* **FROM** students **LIMIT** 1;

Oracle: **SELECT** \* **FROM** students **WHERE** rownum <= 3;

MSSQL Server: **SELECT** **TOP** 3 \* **FROM** students

* If the DB changes or DB design changes then, it will be too expensive to identify the places where changes are required
* JDBC is not good for big applications. There is a big programming overhead.
* We have to manually manage the DB Connection (like Opening & Closing the DB connection / may be via connection pool etc.), takes care of closing of all the JDBC Objects & takes care of mapping the data present in ResultSet to Java beans

**Hibernate**

First do the simple hibernate exercise & through that explain the below concepts

English meaning of Hibernate:

* Hibernate is an open source, **DB Independent,** **Object Relational Mapping (ORM)** framework for **Java** that helps developer to write data handling logic without caring about DB
* Hibernate makes our application code independent of DB. Later if our DB changes or DB Design Changes from the application code point of view we need not to worry much. We just need to worry about the making proper configuration changes
* Hibernate significantly reduce development time & reduces the complex data access code
* Hibernate takes care of mapping Java classes to DB tables & from Java data types to SQL data types
* Hibernate takes care of converting the result set data into java beans. i.e. DTO with hibernate it’s the default behavior
* Developers need have less / no SQL knowledge. With hibernate we need not to worry about writing SQL queries. Hibernate internally takes care of issuing SQL Queries specific to underlying DB
* Maintenance of the application is easier. It’s easier to debug the code & easier to write the error free code

Show the students a normal Java File which as multiple lines SQL queries & explain them the difficulty of with no spaces between the lines

pstmt = con.prepareStatement(

"select \* from students"

+"where regno=?" //Error; No Space between students & where

+"and password=?");

**Recap XML before starting the First Hibernate Program. Specially XSD and/or DTD**

**Steps to Setup the Hibernate Development Environment:**

1. Download the Hibernate 3.6 zip file & extract it to some location

**Download Link:**

1. Copy all the JAR files present under extracted folder to separate folder
2. Download the Hibernate support JAR files (slf4j-simple-1.6.1.jar & mysql-connector-java-5.1.21-bin.jar) & keep it in the above folder
3. Create a Java Project in Eclipse by name “Hibernate”
4. Right click on the Hibernate project, go to “build path” then click on “configure build path”
5. Under “Libraries” tab click on “Add External JARs”
6. Select all the Hibernate & support JARs, click on Open & click on OK

**Steps to Configure Hibernate:**

1. Create a **hibernate.cfg.xml** configuration file under **src** (also called as project class path) & provide all the necessary information (Driver Class, DB URL, User Name & Password etc.) of the underlying DB
2. Create the Java Beans which replicates the underlying DB tables
3. Create the Mapping XML File which maps the Java Bean to DB table & Java Bean properties to DB table columns
4. Add the location of mapping file to hibernate.cfg.xmlfile

**Necessary Steps to work with Hibernate**

1. Load the Hibernate Configuration & Mapping files
2. Build the Session Factory
3. Open the session
4. Operate with DB
5. Flush & Close the session

**Hibernate Example:**

The purpose of a DTD (Document Type Definition) is to define the legal building blocks of an XML document.

The most obvious features offered in XSD that are not available in XML's native Document Type Definitions (DTDs) are namespace awareness and datatypes, that is, the ability to define element and attribute content as containing values such as integers and dates rather than arbitrary text.

**===========================================================**

**<hibernate-configuration>**

**<session-factory>**

**<!—BELOW ARE MANDATORY INFORMATION 🡪**

**<property name=*"connection.url"*></property>**

***connection.driver\_class***

***connection.username***

***connection.password***

**<!—BELOW ARE OPTIONAL INFORMATION 🡪**

***connection.autocommit***

***connection.pool\_size***

***dialect* org.hibernate.dialect.MySQL5Dialect**

***show\_sql***

***format\_sql***

**<mapping resource=*"com/mappingfiles/students.hbm.xml"* />**

**===========================================================**

**<hibernate-mapping>**

**<class name=*"com.jspiders.hibernate.beans.StudentsInfoBean"***

**table=*"students"*>**

**<id name=*"regno"* column=*"regno"* />**

**<property name=*"firstName"* column=*"firstname"* />**

**===========================================================**

**Hibernate.cfg.xml**

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

**<hibernate-configuration>**

**<session-factory>**

**<property name=*"connection.url"*>**

jdbc:mysql://localhost:3306/college

</property>

<property name=*"****connection.driver\_class****"*>

com.mysql.jdbc.Driver

</property>

<property name=*"****connection.username****"*>j2ee</property>

<property name=*"****connection.password****"*>j2ee</property>

<property name=*"****connection.autocommit****"*>true</property>

<property name=*"****connection.pool\_size****"*>

10

</property>

<property name=*"****dialect****"*>

org.hibernate.dialect.MySQL5Dialect

</property>

<property name=*"****show\_sql****"*>true</property>

<property name=*"****format\_sql****"*>true</property>

**<mapping resource**=*"com/mappingfiles/students.hbm.xml"* />

</session-factory>

</hibernate-configuration>

**students.hbm.xml**

<?xml version=*"1.0"* encoding = *"UTF-8"* ?>

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name=*"com.jspiders.hibernate.beans.StudentsInfoBean"* table=*"students"*>

<id name=*"regno"* column=*"regno"* />

<property name=*"firstName"* column=*"firstname"* />

<property name=*"middleName"* column=*"middlename"* />

<property name=*"lastName"* column=*"lastname"* />

</class>

</hibernate-mapping>

**MyFirstHibernate**

//1. Load the Hibernate Configuration & Mapping Files

Configuration config = **new** Configuration();

//Keep the below Files under “src”

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

config.addResource("students\_info.hbm.xml");

//2. Build the Session Factory

SessionFactory factory = config.buildSessionFactory();

//3. Open the Session

Session session = factory.openSession();

//4. Operate With DB

//Insert

StudentBean bean1 = **new** StudentBean();

bean1.setRegno(29);

bean1.setFirstNM("Salman");

bean1.setMiddleNM("NA");

bean1.setLastNM("Khan");

session.save(bean1);

session.update(bean1);

session.delete(bean1);

StudentBean bean

= (StudentBean)session.get(StudentBean.**class**, 25);

//5. Flush & Close the Session

session.flush();

session.close();

**Hibernate Configuration & Mapping Files:**

To interact with DB, hibernate makes use of two types of configuration files

1. Hibernate Configuration File
2. Hibernate Mapping File
3. **Hibernate Configuration File**

* On startup, Hibernate consults this file for its operating properties such as DB connection URL, DB username & password, DB dialect etc.,
* Default file name is hibernate.cfg.xml & default location of this file is project class path (i.e. under src folder)

But, it can have any name & it can be present in anywhere.

* We need must have one configuration file per database i.e. if our application uses more than one database or more than one DB Server to store the data then we need to have those many number of configuration files.
* In other words we can have more than one configuration file depending on the number of databases

**About Dialect:**

* + Hibernate makes use SQL Dialect file to convert Hibernate specific DB calls to native Database SQL calls. It’s a "helper" for Hibernate to communicate with the database in its language. So, Dialect’s makes Hibernate is database independent
  + Depending on the underlying DB we need to set proper dialect related to that database in hibernate configuration file
  + However it’s an optional property. If not provided Hibernate it itself chose the proper dialect file depending on the underlying DB
  + “org.hibernate.dialect” is the package in which the hibernate built-in dialects are present

**Loading Hibernate Configuration File:**

The org.hibernate.cfg.Configuration is

used to build an immutable org.hibernate.SessionFactory. The mappings are compiled from various XML mapping files. The org.hibernate.cfg.Configuration is used to build an immutable org.hibernate.SessionFactory.

* “org.hibernate.cfg.Configuration” is a concrete class present in Hibernate which has a non-static method by name “configure()” which helps us to load the Hibernate Configuration File
* This Class has various overloaded version of configure() method. Which helps us to load the Hibernate Configuration File in different ways
* Following are the different ways to load the hibernate configuration file

**1st Way:**

**public** Configuration configure() **throws** HibernateException

* + This method looks for the configuration file with the default name “hibernate.cfg.xml” under default path i.e. under src folder (class path)
  + If the configuration file has the different name other than “hibernate.cfg.xml” then this method throws an exception
  + If the configuration file has “hibernate.cfg.xml” as file name but if it present somewhere other than class path (within the sub-directories of src folder or may be outside of project) then this method throws an exception

Configuration config = **new** Configuration();

config.configure();

**2nd Way:**

**public** Configuration configure(String resource) **throws** HibernateException

* + This method will look out for the Configuration File with the specified name under src i.e. either directly under src or under subdirectories of src

Configuration config = **new** Configuration();

config.configure("any\_name.xml");

OR

config.configure("com/pkg/any\_name.xml");

**3rd Way:**

**public** Configuration configure(File configFile) **throws** HibernateException

* + This method will look out for the Configuration File with the specified file path i.e. outside of the

project

Configuration config = **new** Configuration();

java.io.File configFileLocation

= **new** java.io.File("E:\\j2ee\\any\_name.xml");

config.configure(configFileLocation);

**4th Way:**

**public** Configuration configure(URL url) **throws** HibernateException

* + This method will look out for the Configuration File present in web server i.e. anywhere in the world

Configuration config = **new** Configuration();

java.net.URL configUrl

= **new** java.net.URL("http://localhost:8080/empApp/any\_name.cfg.xml");

config.configure(configUrl);

**5th Way:**

**public** Configuration configure(org.w3c.dom.Document document)

**throws** HibernateException

1. **Hibernate Mapping File**

* Hibernate is also called as “**O**bject **R**elational **M**apping (ORM)” framework / tool. Hence it expects Java Bean representation for every DB Table
* Hibernate Mapping File, as the name implies, this file describes the mapping between the particular Java Bean to the particular DB table & particular bean property to the particular column of the table. Hibernate refers this mapping file whenever it tries to operate on that corresponding DB table (Give PreparedStatement setXXX() method example)
* DB Table & Mapping file **should be in sync** with regards to DB column type & Java bean property type. In other words Java Bean should be the exact replica of DB table
* Hibernate Mapping File should be a XML file & can have any name but usually the file name will be like “*<table\_name>.*hbm.xml”
* Java Bean used in the mapping file SHOULD have public default constructor (otherwise in case of select operation it fails)
* Each instance of the Java Bean in the mapping file is represented by a row in the configured table
* We must have table definition in the mapping file but we can omit the column definitions in the Mapping file but in that case we should use property name as same as column names
* Hibernate expects Primary Key for every table & hence even if the table does not have Primary Key Hibernate expects id tag in mapping file. i.e. <id> or <composite-id> tag is Mandatory & <property> tag is non-mandatory in mapping file
* Java Bean configured in mapping file should have the configured number of properties in it. Otherwise Hibernate throws an exception (In case of Select Operation; PropertyNotFound Exception). However it can also have other properties in it & in this case, Hibernate does not throw any exception in this case
* Generally we won’t use same Java Bean in more than one mapping file but in hibernate it is possible to have the same Java Bean in more than one mapping file
* We can have different mapping file & Java Bean for the **same table** (Interview Question)
* We can also have one mapping file for entire application as shown below

<hibernate-mapping>

<class name=*"pkg.Bean1"* table=*"Table1"*>

---------

---------

---------

</class>

<class name=*" pkg.Bean2"* table=*" Table2"*>

---------

---------

---------

</class>

</hibernate-mapping>

**About <generator> Tag:**

* This is an optional element comes under <id> tag of the mapping file
* The <generator> tag specifies the class name to be used by hibernate to generate the primary key for new record while inserting the record into DB i.e. the generator informs to the hibernate that, how the primary key value for the new record should be generated
* Hibernate uses different primary key generator algorithms, based on the class we define in the <generator> element & implementation of these algorithms are present in the Hibernate API.
* “org.hibernate.id” is the package in which hibernate Generator classes are present
* Hibernate provides different primary key generator classes and all these classes are implements org.hibernate.id.IdentifierGeneratar Interface
* Hibernate offers various generation strategies some of them are

1. assigned
2. increment
3. identity
4. Many More (native, hilo, uuid, etc.,)

**1. assigned**

* This is the default generator class used by the hibernate
* If generator class is assigned, then the programmer is responsible for providing the primary key information

<id name=*"regno"* column=*"regno"*>

<generator class=*"assigned"* />

</id>

**2. increment**

* If generator class is increment, then the **Hibernate** is responsible for assigning the primary key value **of type long, short or integer only**
* This generator uses the formula “ (Max of PK value in DB) + 1” for generating the PK Value
* if we manually assigned the value for primary key for an object, then hibernate doesn’t considers that value

<id name=*"regno"* type=*"java.lang.Integer"* column=*"regno"*>

<generator class=*"increment"* />

</id>

**3. identity**

* If generator class is identity, then the **DB** is responsible for assigning the primary key value **of type long, short or integer only**
* Compared to identity, increment is slower in operation, because hibernate is responsible for generating the PK value by selecting max of PK Value before inserting new record.

**ALTER** **TABLE** students CHANGE regno regno **INT**(10) **NOT** **NULL** AUTO\_INCREMENT;

<id name=*"regno"* column=*"regno"*>

<generator class=*"identity"* />

</id>

**Loading Hibernate Mapping File:**

* Hibernates Configuration.java provides several ways to load hibernate mapping files. By
* There are multiple ways to load the Hibernate Mapping File

**1st Way:**

Configuration config = **new** Configuration();

config.configure();

config.addResource("com/jspiders/students\_info.hbm.xml");

config.addResource("com/jspiders/guardian\_info.hbm.xml");

config.addResource("com/jspiders/course\_info.hbm.xml");

**2nd Way:**

**In the below case we need to have the mapping file by the same bean name in the same package location**

Configuration config = **new** Configuration();

config.configure();

config.addClass(com.jspiders.hibernate.bean.StudentBean.**class**);

config.addClass(com.jspiders.hibernate.bean.GuardianBean.**class**);

config.addClass(com.jspiders.hibernate.bean.CourseBean.**class**);

Configuration cfg = new Configuration()

.addClass(org.hibernate.auction.Item.class)

.addClass(org.hibernate.auction.Bid.class);

Hibernate will then search for mapping files named /org/hibernate/auction/Item.hbm.xmland /org/hibernate/auction/Bid.hbm.xml in the classpath. This approach eliminates any hardcoded filenames.

**3rd Way:**

Configuration config = **new** Configuration();

config.configure();

config.addFile(**new** File("C:\\Mappings\\course\_info.hbm.xml"));

config.addFile(**new** File("C:\\Mappings\\guardian\_info.hbm.xml"));

config.addFile(**new** File("C:\\Mappings\\students\_info.hbm.xml"));

**4th Way:**

//Below will read the Read all mapping documents from a directory tree. Assumes that any file named \*.hbm.xml is a mapping document.

config.addDirectory(**new** File("C:\\HibernateMappingFiles") );

**5th Way:**

config.addURL(

**new**URL("http://localhost:8080/studentsApp/students\_info.hbm.xml"));

config.addURL(

**new**URL("http://localhost:8080/studentsApp/guardian\_info.hbm.xml"));

config.addURL(

**new** URL("http://localhost:8080/studentsApp/course\_info.hbm.xml"));

**6th Way:**

* If we mention the mapping file location in Hibernate configuration file then while loading the configuration file, hibernate will in turn loads all the Hibernate mapping files

Configuration config = **new** Configuration();

config.configure();

**NOTE:-**

* Hibernate Configuration File can have any name & can be present directly under src, sub-directories of src, outside of project or with in the webserver. In short it can be present anywhere in the world
* There should be One Hibernate Configuration File per DB. If our application uses more than one DB, then we have to have one Hibernate Configuration File for each DB. In short we can also have more than one Hibernate Configuration File
* Hibernate Mapping File can have any name & can be present directly under src, sub-directories of src, outside of project or with in the webserver. In short it can be present anywhere in the world
* We can have either one mapping file per Java Bean to Table mapping OR we can also have ONLY ONE mapping file, which maps all the Java Beans to all the tables. In short we can have One mapping file or More than one mapping files

**About org.hibernate.SessionFactory:**

* SessionFactory is an Object representation of underlying DB & Its behavior is controlled by properties supplied in hibernate configuration file. Hence we load the config file & create the SessionFactory Object only once (Hence it is Immutable Object)
* The org.hibernate.SessionFactory is an Expensive (in debug mode show the time taken to generate SessionFactory Object), Thread Safe & Immutable Object
* Hibernate does allow your application to instantiate more than one org.hibernate.SessionFactory. This is useful if we are using more than one database i.e. one SessionFactory instance per database
* As the name implies SessionFactory is responsible for creating org.hibernate.Session instances.the one & only job of SessionFactory object is to produce Session objects
* SessionFactory is an Interface & by invoking buildSessionfactory() non-static method on Configuration object will create object of type SessionFactory

Configuration config = new Configuration();

config.configure();

SessionFactory sessionFactory = config.buildSessionfactory();

OR

SessionFactory sessionFactory = new Configuration().configure().buildSessionfactory();

**Note:** Failing to invoke configure() method (any version) before invoking buildSessionfactory() method will

create SessionFactory object with in-sufficient information which will throw an exception at runtime

**About org.hibernate.Session:**

* Session Object helps us to interact with underlying DB. This is the central Hibernate API class abstracting the persistence services & consists of all the DB interaction related methods
* Session Object is an inexpensive (in debug mode show the time taken to generate SessionFactory & Session Object), non-thread safe (i.e. single-threaded you cannot share it between threads) object & represents a single unit-of-work with the database i.e. Session object must be created before interacting with DB & must be closed after interacting with DB
* Session is an interface, by invoking openSession() method on SessionFactory object will create an object of type Session

SessionFactory sessionFactory = new Configuration().configure().buildSessionfactory();

Session sess = sessionFactory.openSession();

* By invoking flush() method on session object we force hibernate to execute the SQL Commands on DB
* By invoking close() method on session object will close the Session Object
* Session Object wraps a JDBC java.sql.Connection object & acts as a Factory for org.hibernate.Transaction objects

**public** **class** HibernateUtil

{

**private** **static** **final** SessionFactory *sessionFactory*

= **new** Configuration().configure().buildSessionFactory();

//Private Constructor : To Prevent Object Creation & Inheritence

**private** HibernateUtil() {}

//Public Method : Exposed to Client

**public** **static** Session getSession() {

**return** *sessionFactory*.openSession();

}

}//End of Class

To avoid creating too many sessions ThreadLocal class can be used as shown below to get the current session no matter how many times you make call to the currentSession() method.

Session session = factory.getcurrentSession();

**Interface org.hibernate.Session Methods:**

**To Be Done**

Integer regno = (Integer) session.save(studentsObj);

**Find by Primary Key**

* In enterprise applications, an efficient search mechanism is highly needed.
* Find by PK is a simple mechanism to find an Object by primary key in Hibernate. Hibernate provides a set of different techniques to search a persisted object. They are

1. Invoke **get()** method on Session Object

2. Invoke **load()** method on Session Object

1. **get() Method:**

* Get method hits the database as soon as it is called. So, using the Hibernate Session's get method will always trigger a database hit
* It returns the NULL object if the record does not exists with the given PK

Students student = (Students)session.**get**(Students.**class**, 10);

1. **load() Method:**

* The load method only hits the database when a particular field of the entity is accessed. So, if we use the load method to retrieve an entity, but we never actually access any of the fields of that entity, we never actually hit the database.
* It throws an Exception if the record does not exists with the given PK

Students student = (Students)session.**load**(Students.**class**, 10);

**Mapping a Java Bean in more than one Mapping Files**

* In hibernate it is possible to use same Java Bean in more than one mapping file but in that case we must specify an entity name (entity-name) to distinguish between instances of the same Java Bean while interacting with DB
* By default, hibernate uses class name as the entity name

**public** **class** StudentAndGuardianBean

{

//Common Property for Students\_Info Table & Guardian\_Info Table

**private** **int** regno;

//Students\_Info Table Info

**private** String firstNM;

**private** String middleNM;

**private** String lastNM;

//Guardian\_Info Table Info

**private** String gFirstNM;

**private** String gMiddleNM;

**private** String gLastNM;

}

**Students\_info.hbm.xml file:**

<class

name="**com.jspiders.hibernate.beans.StudentAndGuardianBean**" table="students\_info"

entity-name="studentEntity" >

**Guardian\_info.hbm.xml file:**

<class

name="**com.jspiders.hibernate.beans.StudentAndGuardianBean**" table="guardian\_info"

entity-name="guardianEntity" >

**HibernateSingleBeanExample.java**

StudentAndGuardianBean bean = **new** StudentAndGuardianBean();

bean.setRegno(23);

bean.setFirstNM("Salman");

bean.setMiddleNM("NA");

bean.setLastNM("Khan");

//Save the Record into students\_info

session.save("studentEntity", bean);

StudentAndGuardianBean bean = **new** StudentAndGuardianBean();

bean.setRegno(23);

bean.setFirstNM("Salman");

bean.setMiddleNM("NA");

bean.setLastNM("Khan");

//Save the Record into guardian\_info

session.save("guardianEntity", bean);

**Handling Transactions in Hibernate**

* Transactions handling in Hibernate is similar to handling transactions in JDBC
* In Hibernate begin the transaction by invoking beginTransaction() method on Session object
* Once all work is done, either commit or rollback the tansaction by invoking commit() or rollback() methods on transaction object respectively

**Note:** if we commit the transaction then it is not necessary to flush the transaction because it internally invokes the flush() method

***transaction.flush\_before\_completion***

***transaction.auto\_close\_session***

**Interacting with Multiple DB’s / DB Servers using Hibernate**

* Create a configuration file which contains the properties for each DB i.e. for example, if we need to interact with MySQL DB & as well as Oracle DB then create the following hibernate config files

1. Hibernate.MySQL.cfg.xml
2. Hibernate.Oracle.cfg.xml

* Load these config files in the Java Program which needs to interact with these two DB’s using hibernate
* Create the SessionFactory for each DB’s
* Open each Session using corresponding SesssionFactory Objects

**Hibernate with Annotations**

@Entity

@Table(name="students\_info")

**public** **class** Students3 **implements** Serializable

{

@Id

@GeneratedValue(generator="increment")

@GenericGenerator(name="increment", strategy = "increment")

@Column(name = "regno")

**private** **int** regNo;

@Column(name="firstname")

**private** String firstName;

@Column(name="middlename")

**private** String middleName;

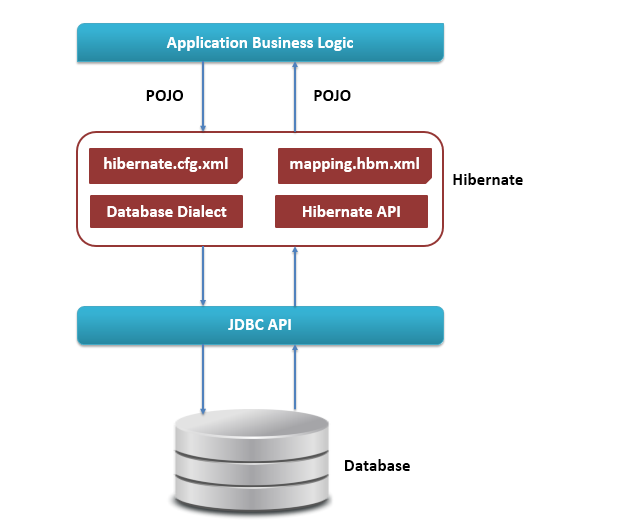
@Column(name="lastname")

**private** String lastName;

}

<mapping class=*"com.jspiders.hibernate.beans.Students3"* />

**Hibernate Architecture:**



**Hibernate Criteria Queries**

* Hibernate provides a functionality by name “Criteria Queries” which helps us to add Zero, One or More restrictions while fetching the records from database
* To make use of Criteria Queries in hibernate, we must create an object of type “org.hibernate.Criteria” by passing the Java Bean class name on which we want to add restrictions

Criteria org.hibernate.Session.createCriteria(Class persistentClass)

* “add()” methods present in Criteria object helps us to add one or more restrictions. To add more than one restrictions we can invoke add() method any number of times

Criteria org.hibernate.Criteria.add(Criterion criterion)

* “org.hibernate.Criterion.Restrictions” has lot of public static methods which helps us to add restrictions to Criteria object
* Invoking “list()” method on Criteria object will instruct the Hibernate to issue appropriate SQL Query & returns the data in the form of “java.util.List”. This list may consist of Zero / More number of objects depending on the Criteria we are adding & the data present in the DB

List org.hibernate.Criteria.list() **throws** HibernateException

* Using Hibernate Criteria Queries, we can put restrictions to ONLY ONE Java Bean at a time

**Ask students write the program including import**

Criteria criteria = session.createCriteria(Students.**class**);

Object[] regno = {10,15,17};

criteria.add(Restrictions.*in*("regno", regno));

criteria.add(Restrictions.*like*("firstName", "varsha"));

criteria.add(Restrictions.between("regno", 5, 10));

criteria.add(Restrictions.like("firstName", "varsha", MatchMode.ANYWHERE));

criteria.add(Restrictions.isNotNull("middleName"));

criteria.add(Restrictions.isNull("middleName"));

criteria.add( Restrictions.like("firstNM", "%s%") );

criteria.addOrder(Order.desc("regno"));

criteria.setMaxResults(10);

List<Students> studentsList = criteria.list();

**for**(Students student : studentsList)

{

System.*out*.println("Reg No : "+student.getRegno());

System.*out*.println("First Name : "+student.getFirstName());

System.*out*.println("Middle Name : "+student.getMiddleName());

System.*out*.println("Last Name : "+student.getLastName());

}

**Managing Associations or Relationships in Hibernate**

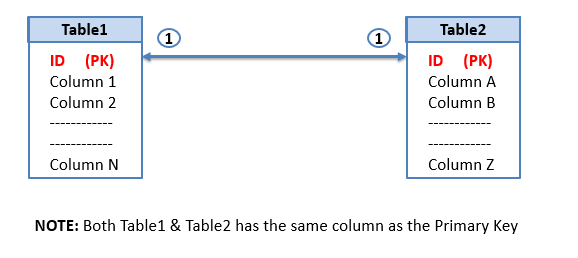
* In DB we can have following types relationships

1. One – to – One
2. One – to – Many
3. Many – to – One
4. Many – to – Many

* Hibernate helps us to handle all the above relationships. In case of Hibernate, we establish the relationship between Java Bean’s, in the corresponding mapping files

1. **Handling One – to – One Relationship in Hibernate**

* One – to – One relationship occurs when one record in “Primary Table” corresponds to exactly one record in the “Relationship Table”
* We can achieve One – to – One relationship between tables by having common column in “Primary Table” & “Relationship Table” having that column as Primary Key in both tables as shown below



* In Hibernate we achieve One – to – One relationship with the help of **<one-to-one>** tag in the mapping file of the Java Bean which corresponds to the Primary Table.

**<one-to-one name=*"propertyNM"***

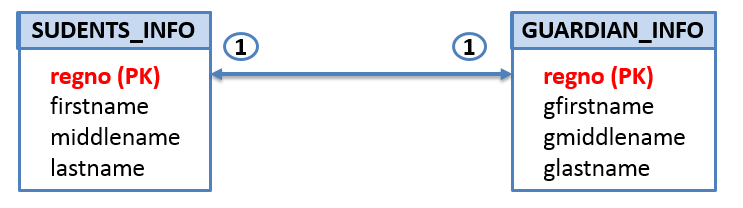
**class=*"pkgNM.RelatedBeanName"***

**cascade=*"all" /*>**

* Cascade attribute present in **<one-to-one>** tag transforms the operation done on the Primary Object to its Related Objects. This attribute has various values,

1. **cascade=*"none"*** (its default) It tells the Hibernate to ignore the relationship. i.e. None of the operation are carried to Related Objects
2. **cascade=*"all"*** All the operations like Save, Update, Delete on Parent Object will be carried to its Related Objects
3. **cascade=*"save"*** Only Save operation on Parent Object will be carried to its Related Objects
4. **cascade=*"update"*** Only Update operation on Parent Object will be carried to its Related Objects
5. **cascade=*"delete"*** Only Delete operation on Parent Object will be carried to its Related Objects
6. **cascade=*"save-update"*** Only Save & Update operation on Parent Object will be carried to its Related Objects
7. Many More (there are other 2)

**Example:**

****

|  |  |
| --- | --- |
| **CREATE** **TABLE** students\_info  ( regno **INT**(50) **NOT** **NULL**,  firstname **VARCHAR**(50),  middlename **VARCHAR**(50),  lastname **VARCHAR**(50),  **PRIMARY** **KEY** (regno)  ); | **CREATE** **TABLE** guardian\_info  ( regno **INT**(50) **NOT** **NULL**,  gfirstname **VARCHAR**(50),  gmiddlename **VARCHAR**(50),  glastname **VARCHAR**(50),  **PRIMARY** **KEY** (regno)  ); |

**Changes done for StudentInfoBean**

//For One-to-One Relationship

**private** GuardianInfoBean guardian;

//Generate Getters & Setters

**Changes done for students\_info.hbm.xml Mapping File**

<one-to-one

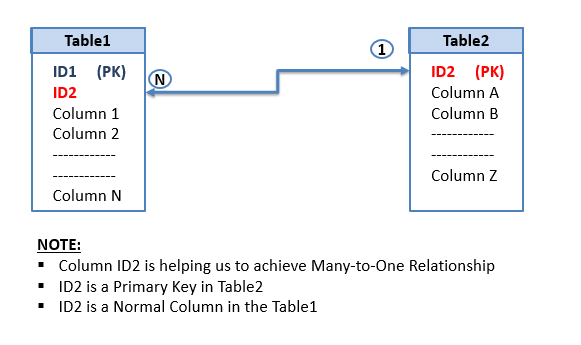
name=*"guardian"*

class=*"com.jspiders.hibernate.beans.GuardianInfoBean"*

cascade=*"all"* />

1. **Handling Many – to – One Relationship in Hibernate**

* Many– to – One relationship occurs when Many records in “Primary Table” corresponds to exactly one record in the “Relationship Table”
* We can achieve Many – to – One relationship between tables by having common column in “Primary Table” & “Relationship Table” having that column as Normal Column in “Primary Table” & Primary Key in “Relationship Table” table as shown below



* In Hibernate we achieve Many – to – One relationship with the help of **<Many-to-one>** tag in the mapping file of the Java Bean which corresponds to the Primary Table.

<many-to-one

name=*"propertyNM"*

column=*"relationshipColumn"*

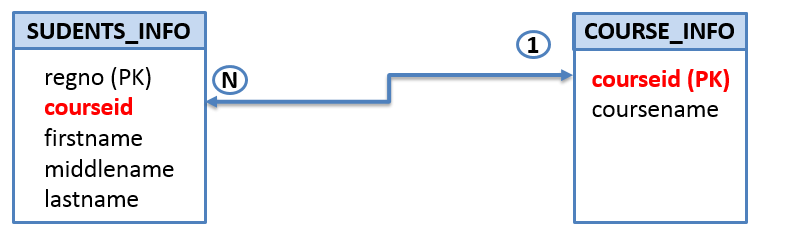
class=*"pkgNM.RelationshipBeanName"*

fetch=*"select"* />

* Fetch attribute instruct the Hibernate how to fetch the Related object records which fetching the Primary Object. It helps Hibernate to generate Select SQL queries based on the value it has.

1. fetch=*"select"* will load all the Related Objects whenever we try to operate with related objects
2. fetch=*"join"* will always load all the Related Objects while reading the Primary Object

**Example:**

****

|  |  |
| --- | --- |
| **ALTER** **TABLE** students\_info  **ADD** **COLUMN** courseid **INT**(10) **NOT** **NULL** **default** '1'; | **CREATE** **TABLE** course\_info  ( courseid **INT**(50) **NOT** **NULL**,  coursename **VARCHAR**(50) ,  **PRIMARY** **KEY** (courseid)  ); |

**Changes done for StudentInfoBean**

//For Many-to-One Relationship

**private** CourseInfoBean course;

//Generate Getters & Setters

**Changes done for students\_info.hbm.xml Mapping File**

<many-to-one name=*"course"*

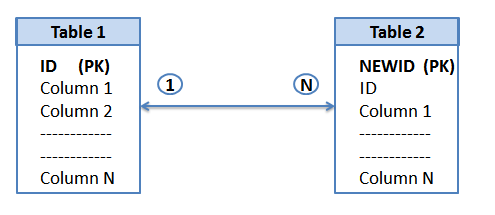
column=*"courseid"*

class=*"com.jspiders.hibernate.beans.CourseInfoBean"*

fetch=*"join"* />

1. **Handling One – to – Many Relationship in Hibernate**

One-to-Many relationship occurs when one record of one table corresponds to multiple records in another table.



One-to-Many relationship occurs when single record of one table corresponds to multiple records in another table.

**Phone\_numbers Table:**

**CREATE** **TABLE** phone\_numbers

( number **INT**(10) **NOT** **NULL**,

regno **INT**(10) **NOT** **NULL**,

number\_type **VARCHAR**(50) **NOT** **NULL**,

**PRIMARY** **KEY** (number, regno)

);

**PhoneNumber Bean**

/\*\*

\* Hibernate requires the class that represents the

\* Composite key must be Serializable

\*

\* **@author** Praveen

\*/

**public** **class** PhoneNumber **implements** Serializable

{

**private** **int** regno;

**private** String numberType;

**private** **int** number;

/\*

\* Generate Getters & Setters

\*/

}

**Phone Number Mapping File**

<hibernate-mapping>

<class name=*"com.jspiders.hibernate.beans.PhoneNumber"* table=*"phone\_numbers"*>

<composite-id>

<key-property name=*"*number*"* column=*"number"*/>

<key-property name=*"regno"* column=*"regno"* />

</composite-id>

<property name=*"* *number\_type"* column=*"number"* />

</class>

</hibernate-mapping>

**Students Bean Changes**

**private** Set<PhoneNumber> phoneNumbers = **new** HashSet<PhoneNumber>();

//Generate Getters & Setters

**Students Mapping File Changes**

<set name=*"phoneNumbers"* table=*"phone\_numbers"* cascade=*"all"* **inverse=*"true"***>

<key column=*"regno"* />

<one-to-many class=*"com.jspiders.hibernate.beans.PhoneNumber"* />

</set>

**Hibernate Config File Changes**

<mapping resource=*"com/jspiders/hibernate/mappingfiles/* *phone\_numbers.hbm.xml"* />

**HibernateOneToManyExample:**

**public** **class** HibernateOneToManyExample

{

**public** **static** **void** main(String[] args)

{

Session session = HibernateUtil.*getNewSession*();

Users user = **null**;

Students student = **null**;

PhoneNumber phoneNumber = **null**;

Course course = **null**;

Set<PhoneNumber> phoneNos = **new** HashSet<PhoneNumber>();

**int** regno=4;

**int** courseId=3;

/\*

\* One-to-Many: Save

\*/

student = **new** Students();

student.setRegno(regno);

student.setFirstName("Suma");

student.setLastName("Sunil");

user = **new** Users();

user.setRegno(regno);

student.setUser(user);

phoneNumber = **new** PhoneNumber();

phoneNumber.setNumberType("Mobile");

phoneNumber.setNumber(123456789);

phoneNumber.setRegno(regno);

phoneNos.add(phoneNumber);

phoneNumber = **new** PhoneNumber();

phoneNumber.setNumberType("Landline");

phoneNumber.setNumber(12345678);

phoneNumber.setRegno(regno);

phoneNos.add(phoneNumber);

phoneNumber = **new** PhoneNumber();

phoneNumber.setNumberType("Fax");

phoneNumber.setNumber(12345);

phoneNumber.setRegno(regno);

phoneNos.add(phoneNumber);

student.setPhoneNumbers(phoneNos);

course = **new** Course();

course.setCourseId(courseId);

student.setCourse(course);

session.save(student);

/\*

\* One-to-Many: Select

\*/

System.*out*.println("Getting the Record from DB");

student = (Students)session.get(Students.**class**, regno);

**if**(student == **null**){

System.*out*.println("Record does not exists ...");

}**else**{

System.*out*.println("Record exists ...");

System.*out*.println("Reg No : "+student.getRegno());

System.*out*.println("First Name : "+student.getFirstName());

System.*out*.println("Middle Name : "+student.getMiddleName());

System.*out*.println("Last Name : "+student.getLastName());

user = student.getUser();

System.*out*.println("Reg No == "+user.getRegno());

System.*out*.println("Password == "+user.getPassword());

System.*out*.println("isadmin : "+user.getIsadmin());

course = student.getCourse();

System.*out*.println("Course ID - "+course.getCourseId());

System.*out*.println("Course Name - "+course.getCourseName());

phoneNos = student.getPhoneNumbers();

Iterator<PhoneNumber> iterator = phoneNos.iterator();

**while**(iterator.hasNext())

{

phoneNumber = iterator.next();

System.*out*.println("Reg No \*\*\* "+phoneNumber.getRegno());

System.*out*.println("No. Type \*\*\* "+phoneNumber.getNumberType());

System.*out*.println("Number \*\*\* "+phoneNumber.getNumber());

}

}

/\*

\* One-to-Many: Update

\*/

user = **new** Users();

user.setRegno(regno);

user.setPassword("asdfg");

student = **new** Students();

student.setRegno(regno);

student.setFirstName("Suma123");

student.setLastName("Sunil123");

student.setUser(user);

course = **new** Course();

course.setCourseId(courseId);

student.setCourse(course);

phoneNumber = **new** PhoneNumber();

phoneNumber.setNumberType("Mobile");

phoneNumber.setNumber(99999999);

phoneNumber.setRegno(regno);

phoneNos.add(phoneNumber);

phoneNumber = **new** PhoneNumber();

phoneNumber.setNumberType("Landline");

phoneNumber.setNumber(9999999);

phoneNumber.setRegno(regno);

phoneNos.add(phoneNumber);

phoneNumber = **new** PhoneNumber();

phoneNumber.setNumberType("Fax");

phoneNumber.setNumber(99999);

phoneNumber.setRegno(regno);

phoneNos.add(phoneNumber);

student.setPhoneNumbers(phoneNos);

session.update(student);

/\*

\* One-to-Many: Delete

\*/

student = **new** Students();

student.setRegno(regno);

user = **new** Users();

user.setRegno(regno);

student.setUser(user);

phoneNumber = **new** PhoneNumber();

phoneNumber.setRegno(regno);

phoneNumber.setNumberType("Mobile");

phoneNos.add(phoneNumber);

phoneNumber = **new** PhoneNumber();

phoneNumber.setRegno(regno);

phoneNumber.setNumberType("Landline");

phoneNos.add(phoneNumber);

phoneNumber = **new** PhoneNumber();

phoneNumber.setRegno(regno);

phoneNumber.setNumberType("Fax");

phoneNos.add(phoneNumber);

student.setPhoneNumbers(phoneNos);

session.delete(student);

session.flush();

session.close();

}

}

**About Inverse Keyword:**

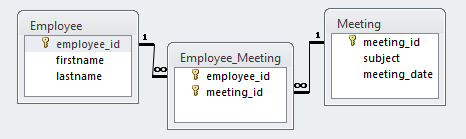
* In Hibernate, only the “relationship owner” should maintain the relationship, and the “inverse” keyword tells which side is the owner to maintain the relationship.
* The “**inverse**” keyword is used with **one-to-many** and **many-to-many** relationships.

many-to-one & one-to-one doesn’t has inverse keyword

* In short, inverse=”true” means the current entity is the relationship owner, and inverse=”false” (default) means it’s not.

1. **Handling Many – to – Many Relationship in Hibernate**

Many-to-Many relationship occurs when Many records of one table corresponds to multiple records in another table.



**DB Changes:**

**CREATE** **TABLE** students\_interview

( regno **INT**(50) **NOT** **NULL**,

interviewid **INT**(50) **NOT** **NULL**,

**PRIMARY** **KEY** (regno,interviewid)

);

**CREATE** **TABLE** interview\_details

( interviewid **INT**(50) **NOT** **NULL**,

companyname **VARCHAR**(50) **NOT** **NULL**,

technology **VARCHAR**(50) **NOT** **NULL**,

location **VARCHAR**(50),

**PRIMARY** **KEY** (interviewid)

);

**InterviewDetails.java**

**public** **class** InterviewDetails

{

**private** **int** interviewID;

**private** String companyNM;

**private** String technology;

**private** String location;

**private** Set<Students> students = **new** HashSet<Students>();

/\*

\* Generate Getters & Setters

\*/

}

**interview\_details.hbm.xml**

<hibernate-mapping>

<class name = *"com.jspiders.hibernate.beans.InterviewDetails"*

Table = *"interview\_details"*>

<id name=*"interviewID"* column=*"interviewid"* />

<property name=*"companyNM"* column=*"companyname"* />

<property name=*"technology"* column=*"technology"* />

<property name=*"location"* column=*"location"* />

<set name=*"students"* table=*"students\_interview"*

inverse=*"true"* lazy=*"true"* lazy=*"true"* cascade=*"all"* >

<key column=*"interviewid"* not-null=*"true"* />

<many-to-many entity-name=*"com.jspiders.hibernate.beans.Students"*

column=*"regno"* />

</set>

</class>

</hibernate-mapping>

**Hibernate.cfg.xml Changes:**

<mapping resource=*"com/jspiders/hibernate/mappingfiles/interviewdetails.hbm.xml"*/>

**StudentsBean.java Changes:**

**private** Set<InterviewDetails> interviewDetails = **new** HashSet<InterviewDetails>();

/\*

\* Generate Getters & Setters

\*/

**Students.hbm.xml Changes:**

<set name=*"interviewDetails"* table=*"students\_interview"*

inverse=*"false"* lazy=*"true"* cascade=*"all"* >

<key column=*"regno"* not-null=*"true"* />

<many-to-many entity-name=*"com.jspiders.hibernate.beans.InterviewDetails"*

column=*"interviewid"* />

</set>

**HibernateManyToMany Example:**

Students student = **null**;

Users user = **null**;

Course course = **null**;

PhoneNumber number = **null**;

InterviewDetails interview = **null**;

Set<PhoneNumber> phoneNos = **new** HashSet<PhoneNumber>();

Set<InterviewDetails> intDetails = **new** HashSet<InterviewDetails>();

**int** regno=9;

**int** courseId=2;

String courseName = "J2EE Design Patterns";

**int** intId = 3;

String compNM = "Capgemini";

String technology = "Core Java";

String location = "Bangalore";

**int** intId2 = 4;

String compNM2 = "Cognizant";

String technology2 = "J2EE";

String location2 = "Bangalore";

/\*

\* Many-to-Many : Save / Update

\*/

student = **new** Students();

//Populate

user = **new** Users();

//Populate

student.setUser(user);

course = **new** Course();

//Populate

student.setCourse(course);

number = **new** PhoneNumber();

//Populate

phoneNos.add(number);

number = **new** PhoneNumber();

//Populate

phoneNos.add(number);

number = **new** PhoneNumber();

//Populate

phoneNos.add(number);

student.setPhoneNumbers(phoneNos);

interview = **new** InterviewDetails();

interview.setInterviewID(intId);

interview.setCompanyNM(compNM);

interview.setLocation(location);

interview.setTechnology(technology);

intDetails.add(interview);

interview = **new** InterviewDetails();

interview.setInterviewID(intId2);

interview.setCompanyNM(compNM2);

interview.setLocation(location2);

interview.setTechnology(technology2);

intDetails.add(interview);

student.setInterviewDetails(intDetails);

session.saveOrUpdate(student);

/\*

\* Many-to-Many : Select

\*/

System.*out*.println("Getting the Record from DB");

student = (Students)session.get(Students.**class**, regno);

**if**(student == **null**){

System.*out*.println("Record does not exists ...");

}**else**{

user = student.getUser();

course = student.getCourse();

phoneNos = student.getPhoneNumbers();

**if**(phoneNos!=**null**)

{

Iterator<PhoneNumber> iterator = phoneNos.iterator();

**while**(iterator.hasNext())

{

number = iterator.next();

}

}

intDetails = student.getInterviewDetails();

**if**(intDetails!=**null**)

{

Iterator<InterviewDetails> iterator2 = intDetails.iterator();

**while**(iterator2.hasNext())

{

interview = iterator2.next();

SOP("Interview ID ###"+interview.getInterviewID());

}

}

}

**Hibernate Query Language (HQL)**

* In SQL, we can write a Single Query to update the multiple columns present in multiple tables
* But we cannot write a Single SQL query which insert data into multiple tables & also ee cannot write a Single SQL query which delete data from multiple tables (So hibernate session.delete() & hibernate session.save() will help in this case but still we can make use of HQL for this as well)

**Possible:**

**update** students\_info si, students\_otherinfo soi

**set** si.lastname='XXX', soi.password='XXX'

**where** si.regno = soi.regno

**and** si.regno = 1;

**Not-Possible:**

**delete** **from** students\_info si, students\_otherinfo soi

**where** si.regno = soi.regno

**and** si.regno = 1;

* Hibernate provides a query language similar to standard SQL to perform operations on the Hibernate Objects. Unlike SQL Queries which operate on DB Tables, HQL operates on the Java Beans
* HQL’s syntax is very similar to standard SQL & the advantage of using HQL over standard SQL is that, SQL is DB Dependent whereas HQL is DB independent
* HQL is case insensitive except the Java Bean class name and its property names. Hence keyword “from” & “FROM” are same in Hibernate whereas “StudentsInfoBean” is not as same as “STUDENTSINFOBEAN”
* We can also use direct package name in the HQL i.e.

String hibernateQuery = "from com.jspiders.hibernate.beans.Students where regno >= :minRegNo";

* To write & execute the HQL we can follow below steps

1. Construct HQL

For ex : String hql = “from JavaBeanName”;

1. Create “org.hibernate.Query” Object

Query query = session.createQuery(hql);

1. Execute the HQL by using appropriate methods on “Query” object
   * Invoke “list()” method is the Query is of type SELECT

List listResult = query.list();

* + Invoke “executeUpdate()” method is the Query is of type INSERT / UPDATE / DELETE

int rowsAffected = query.executeUpdate();

1. Extract result returned from the query depending of the type of the query. For example, “list()” method return java.util.List of Objects & “executeUpdate()” method returns rows affected integer count

**Example:**

/\*

\* HQL : Delete Example

\*/

String deleteQuery = "delete from StudentBean where regno >= :minRegNo";

Query query = session.createQuery(deleteQuery);

query.setParameter("minRegNo", 20);

**int** count = query.executeUpdate();

System.*out*.println("Count : "+count);

/\*

\* HQL : Update Example for Single Table

\*/

String updateQuery

= "update StudentBean set firstNM = :fNM where regno >= :minRegNo";

Query query = session.createQuery(updateQuery);

query.setParameter("fNM", "XXX");

query.setParameter("minRegNo", 5);

**int** count = query.executeUpdate();

System.*out*.println("Count : "+count);

/\*

\* HQL: Insert Example 1. HQL supports INSERT INTO clause only where records

\* can be inserted from one object to another object.

\*/

String insertQuery

= "insert into StudentBean (regno, firstNM, middleNM, lastNM) "

+" SELECT regno, gFirstNM, gMiddleNM,gLastNM FROM GuardianBean ";

Query query = session.createQuery(insertQuery);

**int** count = query.executeUpdate();

System.*out*.println("Count : "+count);

/\*

\* HQL : Insert Example 2

\*/

String insertQuery

= "insert into StudentBean (regno, firstNM, middleNM, lastNM) "

+" SELECT g.regno, g.gFirstNM, g.gMiddleNM,g.gLastNM "

+" FROM GuardianBean g where g.regno = :regNo ";

Query query = session.createQuery(insertQuery);

query.setParameter("regNo", 500);

**int** count = query.executeUpdate();

System.*out*.println("Count : "+count);

/\*

\* HQL : Single Table, ALL columns selection

\*/

//String singleTableQuery = "from StudentBean";

//String singleTableQuery = "from StudentBean where regno >= 5";

//String singleTableQuery = "from StudentBean where firstNM like '111%'";

String singleTableQuery = "from StudentBean where regno >= :minRegNo";

Query query = session.createQuery(singleTableQuery);

query.setParameter("minRegNo", 10);

List<Object> resultList = query.list();

**for**(Object result : resultList)

{

StudentBean student = (StudentBean) result;

System.*out*.println("Reg No : "+student.getRegno());

System.*out*.println("First Name : "+student.getFirstNM());

System.*out*.println("Middle Name : "+student.getMiddleNM());

System.*out*.println("Last Name : "+student.getLastNM());

}

/\*

\* HQL : Single Table, Few columns selection

\*/

String singleTableMultipleFeildsQuery

= "select firstNM, lastNM from StudentBean where regno >= :minRegNo";

Query query = session.createQuery(singleTableMultipleFeildsQuery);

query.setParameter("minRegNo", 10);

List<Object[]> resultList = query.list();

**for**(Object[] result : resultList)

{

String firstName = (String) result[0];

String lastName = (String) result[1];

System.*out*.println("First Name : "+firstName);

System.*out*.println("Last Name : "+lastName);

}

/\*

\* HQL : Multiple Table, ALL columns selection

\*/

String multipleTableQuery

= "from StudentBean s, GuardianBean g " +

" where s.regno = g.regno " +

" and s.regno >= :minRegNo";

Query query = session.createQuery(multipleTableQuery);

query.setParameter("minRegNo", 10);

List<Object[]> resultList = query.list();

**for**(Object[] resultObj : resultList)

{

StudentBean studentObj = (StudentBean)resultObj[0];

GuardianBean guardianObj = (GuardianBean)resultObj[1];

System.*out*.println("Reg No : "+studentObj.getRegno());

System.*out*.println("First Name : "+studentObj.getFirstNM());

System.*out*.println("G First Name : "+guardianObj.getgFirstNM());

System.*out*.println("G Last Name : "+guardianObj.getgLastNM());

}

/\*

\* HQL : Multiple Table, FEW columns selection

\*/

String multipleTableMultipleFeildsQuery

= "select s.regno, s.firstNM, g.gFirstNM " +

" from StudentBean s, GuardianBean g " +

" where s.regno = g.regno " +

" and s.regno >= :minRegNo";

Query query = session.createQuery(multipleTableMultipleFeildsQuery);

query.setParameter("minRegNo", 10);

List<Object[]> resultList = query.list();

**for**(Object[] resultObj : resultList)

{

System.*out*.println("Reg No : "+resultObj[0]);

System.*out*.println("S First Name : "+resultObj[1]);

System.*out*.println("G First Name : "+resultObj[2]);

}

**Hibernate Native SQL Query (HQL)**

Session session = HibernateUtil.*getNewSession*();

/\*

\* HQL : Single Table, ALL columns selection

\*/

String singleTableQuery

= "select \* from students where regno >= :minRegNo";

SQLQuery sqlQuery1 = session.createSQLQuery(singleTableQuery);

sqlQuery1.addEntity(Students.**class**);

sqlQuery1.setParameter("minRegNo", 2);

List<Object> resultList1 = sqlQuery1.list();

**for**(Object result : resultList1)

{

Students student = (Students) result;

System.*out*.println("Reg No : "+student.getRegno());

System.*out*.println("First Name : "+student.getFirstName());

System.*out*.println("Middle Name : "+student.getMiddleName());

System.*out*.println("Last Name : "+student.getLastName());

}

/\*

\* HQL : Single Table, Few columns selection

\*/

String singleTableMultipleFeildsQuery

= "select firstname from students where regno >= :minRegNo";

SQLQuery sqlQuery2 =

session.createSQLQuery(singleTableMultipleFeildsQuery);

sqlQuery2.setParameter("minRegNo", 2);

List<Object> resultList2 = sqlQuery2.list();

**for**(Object result : resultList2)

{

String firstName = (String) result;

System.*out*.println("First Name : "+firstName);

}

/\*

\* HQL : Multiple Table, ALL columns selection

\*/

String multipleTableQuery = "select \* from students s, users u " +

" where s.regno = u.regno " +

" and s.regno >= :minRegNo";

SQLQuery sqlQuery3 = session.createSQLQuery(multipleTableQuery);

sqlQuery3.addEntity(Students.**class**);

sqlQuery3.addEntity(Users.**class**);

sqlQuery3.setParameter("minRegNo", 2);

List<Object[]> resultList3 = sqlQuery3.list();

**for**(Object[] resultObj : resultList3)

{

Students studentObj = (Students)resultObj[0];

Users userObj = (Users)resultObj[1];

System.*out*.println("Reg No : "+studentObj.getRegno());

System.*out*.println("First Name : "+studentObj.getFirstName());

System.*out*.println("Middle Name : "+studentObj.getMiddleName());

System.*out*.println("Last Name : "+studentObj.getLastName());

System.*out*.println("Password : "+userObj.getPassword());

System.*out*.println("isadmin : "+userObj.getIsadmin());

}

/\*

\* HQL : Multiple Table, ALL columns selection

\*/

String multipleTableMultipleFeildsQuery

= "select s.regno, s.middlename, u.isadmin " +

" from students s, users u " +

" where s.regno = u.regno " +

" and s.regno >= :minRegNo";

SQLQuery sqlQuery4

= session.createSQLQuery(multipleTableMultipleFeildsQuery);

sqlQuery4.setParameter("minRegNo", 2);

List<Object[]> resultList4 = sqlQuery4.list();

**for**(Object[] resultObj : resultList4)

{

System.*out*.println("Reg No : "+resultObj[0]);

System.*out*.println("Middle Name : "+resultObj[1]);

System.*out*.println("isadmin : "+resultObj[2]);

}

**public** **class** HibernateNativeSQLQuery

{

**public** **static** **void** main(String[] args)

{

Session session = HibernateUtil.*getNewSession*();

/\*

\* HQL : Single Table, ALL columns selection

\*/

String singleTableQuery = "select \* from students where regno >= :minRegNo";

SQLQuery sqlQuery1 = session.createSQLQuery(singleTableQuery);

sqlQuery1.addEntity(Students.**class**);

sqlQuery1.setParameter("minRegNo", 2);

List<Object> resultList1 = sqlQuery1.list();

**for**(Object result : resultList1)

{

Students student = (Students) result;

System.*out*.println("Reg No : "+student.getRegno());

System.*out*.println("First Name : "+student.getFirstName());

System.*out*.println("Middle Name : "+student.getMiddleName());

System.*out*.println("Last Name : "+student.getLastName());

}

/\*

\* HQL : Single Table, Few columns selection

\*/

String singleTableMultipleFeildsQuery = "select firstname from students where regno >= :minRegNo";

SQLQuery sqlQuery2 = session.createSQLQuery(singleTableMultipleFeildsQuery);

sqlQuery2.setParameter("minRegNo", 2);

List<Object> resultList2 = sqlQuery2.list();

**for**(Object result : resultList2)

{

String firstName = (String) result;

System.*out*.println("First Name : "+firstName);

}

/\*

\* HQL : Multiple Table, ALL columns selection

\*/

String multipleTableQuery = "select \* from students s, users u " +

" where s.regno = u.regno " +

" and s.regno >= :minRegNo";

SQLQuery sqlQuery3 = session.createSQLQuery(multipleTableQuery);

sqlQuery3.addEntity(Students.**class**);

sqlQuery3.addEntity(Users.**class**);

sqlQuery3.setParameter("minRegNo", 2);

List<Object[]> resultList3 = sqlQuery3.list();

**for**(Object[] resultObj : resultList3)

{

Students studentObj = (Students)resultObj[0];

Users userObj = (Users)resultObj[1];

System.*out*.println("Reg No : "+studentObj.getRegno());

System.*out*.println("First Name : "+studentObj.getFirstName());

System.*out*.println("Middle Name : "+studentObj.getMiddleName());

System.*out*.println("Last Name : "+studentObj.getLastName());

System.*out*.println("Password : "+userObj.getPassword());

System.*out*.println("isadmin : "+userObj.getIsadmin());

}

/\*

\* HQL : Multiple Table, ALL columns selection

\*/

String multipleTableMultipleFeildsQuery =

"select s.regno, s.middlename, u.isadmin " +

" from students s, users u " +

" where s.regno = u.regno " +

" and s.regno >= :minRegNo";

SQLQuery sqlQuery4 =

session.createSQLQuery(multipleTableMultipleFeildsQuery);

sqlQuery4.setParameter("minRegNo", 2);

List<Object[]> resultList4 = sqlQuery4.list();

**for**(Object[] resultObj : resultList4)

{

System.*out*.println("Reg No : "+resultObj[0]);

System.*out*.println("Middle Name : "+resultObj[1]);

System.*out*.println("isadmin : "+resultObj[2]);

}

}//End of Main

}//End of Class

MySQL store procedure

Here’s a MySQL store procedure, which accept a stock code parameter and return the related stock data.

DELIMITER $$

**CREATE** **PROCEDURE** `GetStocks`(int\_stockcode **VARCHAR**(20))

**BEGIN**

**SELECT** \* **FROM** stock **WHERE** stock\_code = int\_stockcode;

**END** $$

DELIMITER ;

In MySQL, you can simple call it with a **call** keyword :

**CALL** GetStocks('7277');

Hibernate call store procedure

In Hibernate, there are three approaches to call a database store procedure.

1. Native SQL – createSQLQuery

You can use **createSQLQuery()** to call a store procedure directly.

Query query = session.createSQLQuery(

"CALL GetStocks(:stockCode)")

.addEntity(Stock.**class**)

.setParameter("stockCode", "7277");

List result = query.list();

**for**(**int** i=0; i<result.size(); i++){

Stock stock = (Stock)result.get(i);

System.out.println(stock.getStockCode());

}