HIBERNATE

**HIBERNATE Framework over JDBC API**

* In JDBC, if we open a database connection we need to write in try, and if any exceptions occurred catch block will takers about it, and finally used to close the connections.
* here as a programmer we must close the connection, or we may get a chance to get our of connections message…!
* Actually if we didn’t close the connection in the finally block, then jdbc doesn’t responsible to close that connection.
* In JDBC we need to write Sql commands in various places, after the program has created if the table structure is modified then the JDBC program doesn’t work, again we need to modify and compile and re-deploy required, which is tedious.
* JDBC used to generate database related error codes if an exception will occurs, but java programmers are unknown about this error codes right.
* In the Enterprise applications, the data flow with in an application from class to class will be in the form of objects, but while storing data finally in a database using JDBC then that object will be converted into text.  Because JDBC doesn’t transfer objects directly.

In order to overcome above problems, Hibernate came into picture..!

**What is Hibernate:**

* Hibernate is a pure Java object-relational mapping (ORM) and persistence framework that allows you to map plain old Java objects to relational database tables.The main goal of hibernate is to relieve the developer from the common data persistence related tasks.It maps the objects in the java with the tables in the database very efficiently and also you can get maximum using its data query and retrieval facilities.

**What Hibernate Does**

* Map Java class to database tables & vice versa
* Data query and retrieval facility
* Generate the SQL query based on the underline DB. and attempts to relieve the developer from manual result set handling and object conversion.
* Make application portable to all relational DB.
* Enhance performance by providing the different levels of cache(First, Second and Query level).

**Hibernate Architecture**

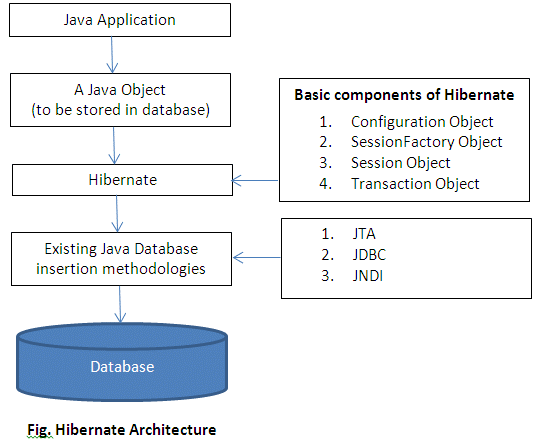


Fig - Detailed view of Hibernate Architecture

Core Interfaces of Hibernate:

a. Configuration Object

* Configuration object is created to load the configuration files like hibernate.proerties, hibernatate.cfg.xml and hbm files (Java  objects to database mapping). Since this step is loading of configurations, it generally happens at the application initialization time.
* We can create a configuration object as shown below

          Configuration configuration = new Configuration().configure();

* New Configuration() call will load the hibernate.properties fileand calling configure() method on configuration object loads hibernate.cfg.xml. In case any property is defined in both hibernate.properties and hibernate. cfg. xml file, then xml file will get precedence.
* To load the custom files , we can call

          new Configuration().configure("/configurations/myConfiguration.cfg.xml")

* The above code snippet will load myConfiguration.cfg.xml  from the  configuration subdirectory of the class path.
* Hibernate does provide an option to load the configurations programmatically as well. To load hbm files we can use addClass()or addResource() method on the configuration object.Database configurations can be loaded using setProperties() method call.

### b. Session Factory

Session Factory can be obtained from Configuration object and is a heavyweight object and can be used with multiple threads as, the session factory object is thread safe.

SessionFactory class is available under org.hibernate.SessionFactory package.

We would always need one instance of Session factory per database that our application is interacting with. So if we have two different databases, we would create two session factory objects.

As Session factory is a heavy weight object, creation of the session factory object is an expensive operation and recommended to get it created at application start up or can be accessed using JNDI.

Use below code snippet to get a Session Factory object.

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

*SessionFactory sessionFactory = configuration.buildSessionFactory();*

Once the session factory object is obtained, configuration object is no longer required.

### c. Sessions

Session objects are created using Session factory and are a lightweight object. Session objects provide a connection with a relational database.

By design, we should create a new session object when a database interaction is needed. A session object represents a unit of work.

Session objects are not thread safe and hence should not keep open for a long time.

Use below code snippet to open a new Session object

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

           SessionFactory sessionFactory = configuration.buildSessionFactory();

  Session session = sessionFactory.openSession();

*Use* session.close() *to close the session.*

We can pass an optional connection object as an argument to openSession() method if we would want to interact with the database using associated connection. However, this approach is not recommended.

### D.Transactions

A transaction is a unit of work in which either all operations must execute or none of them.  The transaction is a short lived and a single threaded object.

Transaction is available under org.hibernate.Transaction package

There are four important terms related to transactions are –

* Atomic -  As described above, atomicity makes sure that either all operations within a transaction must be successful or none of them.
* Consistent - This property makes sure that data should be in consistent state once the transaction is completed.
* Isolated-  this property allows multiple users to access the same set of data and each user’s processing should be isolated from others.
* Durable – Result of the transaction should be permanent once the transaction is completed to avoid any loss of data.

**List of generators**

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

* assigned
* increment
* sequence
* identity

**assigned**

* This generator supports in all the databases
* This is the default generator class used by the hibernate, if we do not specify <generator –> element under id element then hibernate by default assumes it as “assigned”
* 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

**Increment**

* This generator supports in all the databases, database independent
* This generator is used for generating the id value for the new record by using the formula

Max of id value in Database + 1

**sequence**

* Not has the support with MySql
* This generator class is database dependent it means, we cannot use this generator class for all the database, we should know whether the database supports sequence or not before we are working with it
* 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

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

<generator>

<param name="sequence">MySequence</param>

</genetator>

</id>

* If the programmer has not passed any sequence name, then hibernate creates its own sequence with name “**Hibernate-Sequence**” and gets next value from that sequence, and than assigns that id value for new record

But remember, if hibernate want’s to create its own sequence, in hibernate configuration file, **hbm2ddl.auto** property must be set enabled

**identity**

* This is database dependent, actually its not working in oracle

In this case (identity generator) the id value is generated by the database, but not by the hibernate, but in case of increment hibernate will take over this.

Hibernate Query Language Introduction

So far we done the operations on single object (single row), here we will see modifications, updates on multiple rows of data (multiple objects) at a time.  In hibernate we can perform the operations on a single row (or) multiple rows at a time, if we do operations on multiple rows at once, then we can call this as bulk operations.

* HQL is the own query language of hibernate and it is used to perform bulk operations on hibernate programs
* An object oriented form of SQL is called HQL
* here we are going to replace table column names  with POJO class variable names and table names with POJO class names in order to get HQL commands

**Advantages Of HQL:**

* 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 with out doing any further changes to it
* HQL supports object oriented features like ***Inheritance***, ***polymorphism***, ***Associations***(Relation ships)
* HQL is initially given for selecting object from database and in hibernate 3.x we can do DML operations ( insert, update…) too

**Different Ways Of Construction HQL Select**

* If we want to select a Complete Object from the database, we use POJO class reference in place of   **\***  while constructing the query
* In this case (select a complete object from the database) we can directly start our HQL command from,  **from** key word

|  |
| --- |
| // In SQL  sql> select \* from Product  Note: Product is the table name right....!!!    // In HQL  hql> select p from Product p       [ or ]       from Product p  Note: here p is the reference...!! |

If we want to load the Partial Object from the database that is only selective properties (selected columns) of an objects then we need to replace column names with POJO class variable names.

Example:

|  |  |
| --- | --- |
|  | // In SQL  sql> select pid,pname from Product  Note: pid, pname are the columns Product is the table name right..!    // In HQL  hql> select p.productid,p.productName from Product p       [ or ]       from Product p ( we should not start from, from key word here because  we selecting the columns hope you are getting me )    Note: here p is the reference...!!             productid,productName are POJO variables |

* It is also possible to load or select the object from the database by passing runtime values into the query,  in this case we can use either ” ? ” symbol or label in an HQL command, the index number of  ” ? ” will starts from zero but not one ( Remember this, little important regarding interview point of view)

|  |  |
| --- | --- |
|  | // In SQL  sql> select \* from Product where pid=?  Note: Product is the table name right..!    // In HQL  hql> select p from Product p where p.productid=?       [ or ]       select p from Product p where p.productid=:java       [ or ]       from Product p where p.productid=?       [ or ]       from Product p where p.productid=:java    Note: Here p is the reference...!! |

Let us see, how to execute HQL commands..

**Procedure To Execute HQL Command:**

* If we want to execute execute an HQL query on a database, we need to create a query object
* ” Query ” is an interface given in org.hibernate package
* In order to get query object, we need to call createQuery() method in the session Interface
* Query is an interface, QueryImpl is the implemented class
* we need to call list method for executing an HQL command on database, it returns java.util.List
* we need to use java.util.Iterator for iterating the List collection

Syntax:

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

**Notes:**

* line 1: Getting the Query object
* line 2: Executing the object content (which is HQL command)

line 3: Iterating, then while loop and type cast into our class type that’s it

We can execute our HQL command in 3 ways,  like by selecting total object, partial object (more than one column), partial object (with single column).  Let us see..

**Different Ways Of Executing HQL**

**Case 1: [ Selecting Complete Object ]**

* In this approach, we are going to select complete object from the database, so while iterating the collection, we need to typecast each object into our  POJO class type only
* Internally hibernate converts each row selected from the table into an object of POJO class and hibernate stores all these POJO class objects into list so while iterating the collection, we typecast into POJO class type

|  |
| --- |
| Query qry = session.createQuery("-- our HQL command --");  List l =qry.list();  Iterator it = l.iterator();  while(it.hasNext())  {     Object o = it.next();     Project p = (Product)o;     ----- ------- ------  } |

Case 2: [ Selecting Partial Object ]

* In this approach we are going to select partial object, (selected columns, i mean more than one column not single column)
* In this case hibernate internally stores the multiple column values of each row into an object array and stores these object arrays into List collection
* At the time of iterating the collection, we need to typecast the result into an object arrays

|  |  |
| --- | --- |
|  | Query qry = session.createQuery("select p.pid,p.pname from Product p");  List l =qry.list();  Iterator it = l.iterator();  while(it.hasNext())  {     Object o[] = (Object o[])it.next();  System.out.println("Product id : "+o[0]+ "Product Name : "+o[1]);     System.out.println("-------");     ----- ------- ------  } |

* Case 3: [ Selecting Partial Object ]
* In this case we are going to select partial object with single column from the database
* In this case hibernate internally creates an object of that value type and stores all these objects into the list collection
* At the time of iterating the collection, we need to typecast into that object type only

|  |  |
| --- | --- |
|  | Query qry = session.createQuery("select p.pid from Product p");  List l =qry.list();  Iterator it = l.iterator();  while(it.hasNext())  {     Integer i = (Integer)it.next();     System.out.println(i.intValue());     ----- ------- ------  } |

Note:  it.next() return type is always **Object**

Let us see the program on HQL select command,  which is going to cover complete object, partial object (More than one column), partial object (Single column)

here are the required files….

* Product.java (POJO class)
* Product.hbm.xml  (Xml mapping file )
* hibernate.cfg.xml  (Xml configuration file)
* TestLogic.java (java file to write our hibernate logic)

Product.java

|  |  |
| --- | --- |
|  | package str;    public class Product{    private int productId;  private String proName;  private double price;    public void setProductId(int productId)  {      this.productId = productId;  }  public int getProductId()  {      return productId;  }    public void setProName(String proName)  {      this.proName = proName;  }  public String getProName()  {      return proName;  }    public void setPrice(double price)  {      this.price = price;  }  public double getPrice()  {      return price;  }  } |

**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="connection.driver\_class">oracle.jdbc.driver.OracleDriver  </property>  <property name="connection.url">jdbc:oracle:thin:localhost:1521:XE</property>  <property name="connection.username">system</property>  <property name="connection.password">admin</property>    <property name="dialect">org.hibernate.dialect.OracleDialect</property>  <property name="show\_sql">true</property>  <property name="hbm2ddl.auto">update</property>   <mapping resource="Product.hbm.xml"></mapping>  </session-factory>  </hibernate-configuration> |

**Product.hbm.xml**

|  |  |
| --- | --- |
|  | <?xml version="1.0"?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">    <hibernate-mapping>  <class name="str.Product" table="products">    <id name="productId" column="pid"  />  <property name="proName" column="pname" length="10"/>  <property name="price"/>    </class>  </hibernate-mapping> |

TestLogic.java

|  |  |
| --- | --- |
|  | package str;    import org.hibernate.\*;  import org.hibernate.cfg.\*;  import java.util.\*;    public class TestLogic {    public static void main(String[] args)  {    Configuration cfg = new Configuration();  cfg.configure("hibernate.cfg.xml");    SessionFactory factory = cfg.buildSessionFactory();  Session session = factory.openSession();    /\* Selecting all objects(records) start\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \*/    /\*    Query qry = session.createQuery("from Product p");    List l =qry.list();  System.out.println("Total Number Of Records : "+l.size());  Iterator it = l.iterator();    while(it.hasNext())  {  Object o = (Object)it.next();  Product p = (Product)o;  System.out.println("Product id : "+p.getProductId());  System.out.println("Product Name : "+p.getProName());  System.out.println("Product Price : "+p.getPrice());  System.out.println("----------------------");  }    \*/    /\* Selecting all objects(records) end\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \*/    /\* Selecting partial objects(More than one object) start\_\_\_\_\_\_\_\_\_\_ \*/    /\*    Query qry = session.createQuery("select p.productId,p.proName from Product p");    List l =qry.list();  System.out.println("Total Number Of Records : "+l.size());  Iterator it = l.iterator();    while(it.hasNext())  {  Object o[] = (Object[])it.next();    System.out.println("Product id : "+o[0]+ "Product Name : "+o[1]);    System.out.println("----------------");  }    \*/    /\* Selecting partial objects(More than one object)end\_\_\_\_\_\_\_\_\_\_\_\_\_ \*/    // Selecting single object start\_\_\_\_\_\_\_\_\_\_\_\_\_    Query qry = session.createQuery("select p.productId  from Product p");    List l =qry.list();  System.out.println("Total Number Of Records : "+l.size());  Iterator it = l.iterator();    while(it.hasNext())  {  Integer i = (Integer)it.next();  System.out.println("Product id : "+i.intValue());  System.out.println("---------------------------");    }    // selecting single object end\_\_\_\_\_\_\_\_\_\_\_\_    session.close();  factory.close();  }    } |

* Notes regarding TestLogic.java:
* **CASE 1**:  For selecting complete objects (all rows from the table), see from line number 18 – 40 [ just remove the comments ] and check the out put
* **CASE 2**:  For selecting partial objects, i mean more than one object, see from line number 42 – 63 [ just remove the comments ] and check the out put
* **CASE 3**:  For selecting partial objects,  (selecting single row), see from line number 65 – 82 [ just remove the comments ] and check the out put

See the following output just am giving for **CASE 3**, and remember in case1 i typecast into POJO class type, in case 2 typecast into objects array, in case 3 typecast into that value type\_\_ (in the while loop..)

Now we will see, how to pass the values at time time while using the HQL  select query, actually same concept for 3 cases.

Required files…

* Product.java (POJO class)
* Product.hbm.xml  (Xml mapping file )
* hibernate.cfg.xml  (Xml configuration file)
* TestLogic.java (java file to write our hibernate logic)

Product.java

package str;

public class Product{

private int productId;

private String proName;

private double price;

public void setProductId(int productId)

{

this.productId = productId;

}

public int getProductId()

{

return productId;

}

public void setProName(String proName)

{

this.proName = proName;

}

public String getProName()

{

return proName;

}

public void setPrice(double price)

{

this.price = price;

}

public double getPrice()

{

return price;

}

}

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="connection.driver\_class">oracle.jdbc.driver.OracleDriver  </property>  <property name="connection.url">jdbc:oracle:thin:localhost:1521:XE</property>  <property name="connection.username">system</property>  <property name="connection.password">admin</property>   <property name="dialect">org.hibernate.dialect.OracleDialect</property>  <property name="show\_sql">true</property>  <property name="hbm2ddl.auto">update</property>    <mapping resource="Product.hbm.xml"></mapping>  </session-factory>  </hibernate-configuration> |

Product.hbm.xml:

|  |  |
| --- | --- |
|  | <?xml version="1.0"?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">   <hibernate-mapping>  <class name="str.Product" table="products">   <id name="productId" column="pid"  />  <property name="proName" column="pname" length="10"/>  <property name="price"/>   </class>  </hibernate-mapping> |

TestLogic.java

|  |
| --- |
| package str;  import org.hibernate.\*;  import org.hibernate.cfg.\*;  import java.util.\*;  public class TestLogic {  public static void main(String[] args)  {  Configuration cfg = new Configuration();  cfg.configure("hibernate.cfg.xml");  SessionFactory factory = cfg.buildSessionFactory();  Session session = factory.openSession();  Query qry = session.createQuery("from Product p where p.productId= :java");  qry.setParameter("java",105);  List l =qry.list();  System.out.println("Total Number Of Records : "+l.size());  Iterator it = l.iterator();  while(it.hasNext())  {  Object o = (Object) it.next();  Product p = (Product)o;  System.out.println("Product Name : "+p.getProName());  System.out.println("Product Price : "+p.getPrice());  System.out.println("---------------------------");  }  \*/  /\* Using Question Mark  \*/  Query qry = session.createQuery("from Product p where p.productId= ?");   qry.setParameter(0,105);  List l =qry.list();  System.out.println("Total Number Of Records : "+l.size());  Iterator it = l.iterator();  while(it.hasNext())  {  Object o = (Object) it.next();  Product p = (Product)o;  System.out.println("Product Name : "+p.getProName());  System.out.println("Product Price : "+p.getPrice());  System.out.println("---------------------------");  }  session.close();  factory.close();  }   } |

 so far we have been executed the programs on Hibernate Query Language (HQL) select only, now we will see the DML operations in HQL like insert, delete, update., you know some thing..? this delete, update query’s are something similar to the select query only but insert query in Hibernate Query Language(HQL) is quite different, i will let you know while we are going to execute the program on HQL insert, for now let us see an example on delete, update queries

Remember.., while we are working with DML operations in HQL we have to call executeUpdate(); to execute the query, which will returns one integer value after the execution, will discuss more on this later..

files required.. (for both delete, update query’s)

* Product.java (POJO class)
* Product.hbm.xml
* hibernate.cfg.xml
* TestLogic.java (Our logic)

**Let Us See An Example On HQL DELETE**

Actually Product.java, Product.hbm.xml and hibernate configuration files are some for both logics

Product.java

package str;

public class Product{

private int productId;

private String proName;

private double price;

public void setProductId(int productId)

{

this.productId = productId;

}

public int getProductId()

{

return productId;

}

public void setProName(String proName)

{

this.proName = proName;

}

public String getProName()

{

return proName;

}

public void setPrice(double price)

{

this.price = price;

}

public double getPrice()

{

return price;

}

}

Product.hbm.xml

<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC

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

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="str.Product" table="products">

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

<property name="proName" column="pname" length="10"/>

<property name="price"/>

</class>

</hibernate-mapping>

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="connection.driver\_class">oracle.jdbc.driver.OracleDriver  </property>  <property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>  <property name="connection.username">system</property>  <property name="connection.password">admin</property>   <property name="dialect">org.hibernate.dialect.OracleDialect</property>  <property name="show\_sql">true</property>  <property name="hbm2ddl.auto">update</property>  <mapping resource="Product.hbm.xml"></mapping>  </session-factory>  </hibernate-configuration> |

TestLogic.java

|  |
| --- |
| package str;  import org.hibernate.\*;  import org.hibernate.cfg.\*;   public class TestLogic {  public static void main(String[] args)  {  Configuration cfg = new Configuration();  cfg.configure("hibernate.cfg.xml");  SessionFactory factory = cfg.buildSessionFactory();  Session session = factory.openSession();   Query qry = session.createQuery("delete from Product p  where p.productId=:java");          qry.setParameter("java",110);          int res = qry.executeUpdate();          System.out.println("Command successfully executed....");          System.out.println("Numer of records effected due to delete query"+res);  session.close();  factory.close();  }    } |

**Using Named Parameters**

Hibernate supports named parameters in its HQL queries. This makes writing queries that accept input from the user easy—and you do not have to defend against SQL injection attacks.

When using JDBC query parameters, any time you add, change, or delete parts of the SQL statement, you need to update your Java code that sets its parameters, because the parameters are indexed based on the order in which they appear in the statement. Hibernate lets you provide names for the parameters in the HQL query, so you do not have to worry about accidentally moving parameters around in the query.

The simplest example of named parameters uses regular SQL types for the parameters:

|  |  |
| --- | --- |
|  | String hql = "from Product where price > :price";  Query query = session.createQuery(hql);  query.setDouble("price",25.0);  List results = query.list(); |

**Paging Through the Result Set**

Pagination through the result set of a database query is a very common application pattern. Typically, you would use pagination for a web application that returned a large set of data for a query. The web application would page through the database query result set to build the appropriate page for the user. The application would be very slow if the web application loaded all of the data into memory for each user. Instead, you can page through the result set and retrieve the results you are going to display one chunk at a time.

There are two methods on the Query interface for paging: setFirstResult() and setMaxResults(). The setFirstResult() method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to only retrieve a fixed number of objects with the setMaxResults() method. Your HQL is unchanged—you need only to modify the Java code that executes the query.

|  |  |
| --- | --- |
|  | Query query = session.createQuery("from Product");  query.setFirstResult(1);  query.setMaxResults(2);  List results = query.list();  displayProductsList(results); |

If you turn on SQL logging, you can see which SQL commands Hibernate uses for pagination. For the open-source HSQLDB database, Hibernate uses top and limit. Microsoft SQL Server does not support the limit command, so Hibernate uses only the top command. If your application is having performance problems with pagination, this can be very helpful for debugging.

If you only have one result in your HQL result set, Hibernate has a shortcut method for obtaining just that object as discussed next.

**Obtaining a Unique Result**

HQL’s Query interface provides a uniqueResult() method for obtaining just one object from an HQL query. Although your query may yield only one object, you may also use the uniqueResult() method with other result sets if you limit the results to just the first result. You could use the setMaxResults() method discussed in the previous section.

The uniqueResult() method on the Query object returns a single object, or null if there are zero results. If there is more than one result, then the uniqueResult() method throws a NonUniqueResultException.

|  |  |
| --- | --- |
|  | String hql = "from Product where price>25.0";  Query query = session.createQuery(hql);  query.setMaxResults(1);  Product product = (Product) query.uniqueResult(); |

**Sorting Results with the order by Clause**

To sort your HQL query’s results, you will need to use the order by clause. You can order the results by any property on the objects in the result set: either ascending (asc) or descending (desc). You can use ordering on more than one property in the query, if you need to. A typical HQL query for sorting results looks like this:

|  |  |
| --- | --- |
|  | from Product p where p.price>25.0 order by p.price desc |

If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas. For instance, you could sort by product price and the supplier’s name, as follows:

|  |  |
| --- | --- |
|  | from Product p order by p.supplier.name asc, p.price asc |

**Associations**

Associations allow you to use more than one class in an HQL query, just as SQL allows you to use joins between tables in a relational database. You add an association to an HQL query with the join clause. Hibernate supports five different types of joins: **inner join, cross join, left outer join, right outer join, and full outer join**. If you use cross join, just specify both classes in the from clause (from Product p, Supplier s). For the other joins, use a join clause after the from clause. Specify the type of join, the object property to join on, and an alias for the other class.

You can use inner join to obtain the supplier for each product, and then retrieve the supplier name, product name, and product price, as so:

|  |  |
| --- | --- |
|  | select s.name, p.name, p.price from Product p inner join p.supplier as s |

You can retrieve the objects using similar syntax:

|  |  |
| --- | --- |
|  | from Product p inner join p.supplier as s |

**Aggregate Methods**

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL, so you do not have to learn any specific Hibernate terminology. The difference is that in HQL, aggregate methods apply to the properties of persistent objects. You may use the count(\*) syntax to count all the objects in the result set, or count(product.name) to count the number of objects in the result set with a name property. Here is an example using the count(\*) method to count all products:

|  |  |
| --- | --- |
|  | select count(\*) from Product product |

Criteria Query, Hibernate Criteria Query Introduction

Unlike HQL, Criteria is only for selecting the data from the database, that to we can select complete objects only not partial objects, in fact by combining criteria and projections concept we can select partial objects too we will see this angle later, but for now see how we are using criteria for selecting complete objects form the database. We cant perform non-select operations using this criteria.  Criteria is suitable for executing dynamic queries too, let us see how to use this criteria queries in the hibernate..

Syntax:

Criteria crit = session.createCriteria(–Our class object–);

|  |  |
| --- | --- |
|  | Criteria crit = session.createCriteria(Employee.class);  // let Product is our pojo class  List l = crit.list()  // need to call list() to execute criteria  Iterator it = l.iterator();  while(it.hasNext())  {  Object o = it.next();  Product p = (Product)o;  ------ ----- -----  } |

Examples of Hibernate Criteria Query Language

There are given a lot of examples of HCQL.

Example of HCQL to get all the records

1. Crietria c=session.createCriteria(Emp.class);//passing Class class argument
2. List list=c.list();

Example of HCQL to get the 10th to 20th record

1. Crietria c=session.createCriteria(Emp.**class**);
2. c.setFirstResult(10);
3. c.setMaxResult(20);
4. List list=c.list();

Example of HCQL to get the records whose salary is greater than 10000

Crietria c=session.createCriteria(Emp.class);

c.add(Restrictions.gt("salary",10000));//salary is the propertyname

List list=c.list();

Example of HCQL to get the records in ascending order on the basis of salary

1. Crietria c=session.createCriteria(Emp.**class**);
2. c.addOrder(Order.asc("salary"));
3. List list=c.list();

## Using Restrictions with Criteria

The Criteria API makes it easy to use restrictions in your queries to selectively retrieve objects; for instance, your application could retrieve only products with a price over $30. You may add these restrictions to a Criteria object with the add() method. The add() method takes an org.hibernate.criterion.Criterion object that represents an individual restriction. You can have more than one restriction for a criteria query.

**i) Restrictions.eq() Example**

To retrieve objects that have a property value that “**equals**” your restriction, use the eq() method on Restrictions, as follows:

|  |  |
| --- | --- |
|  | Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.eq("description","Mouse"));  List<Product> results = crit.list() |

Above query will search all products having description as “Mouse”.

**ii) Restrictions.ne() Example**

To retrieve objects that have a property value “not equal to” your restriction, use the ne() method on Restrictions, as follows:

|  |  |
| --- | --- |
|  | Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.ne("description","Mouse"));  List<Product> results = crit.list() |

Above query will search all products having description anything but not “Mouse”.

You cannot use the not-equal restriction to retrieve records with a NULL value in the database for that property (in SQL, and therefore in Hibernate, NULL represents the absence of data, and so cannot be compared with data). If you need to retrieve objects with NULL properties, you will have to use the isNull() restriction.

**iii) Restrictions.like() and Restrictions.ilike() Example**

Instead of searching for exact matches, we can retrieve all objects that have a property matching part of a given pattern. To do this, we need to create an SQL LIKE clause, with either the like() or the ilike() method. The ilike() method is case-insensitive.

|  |  |
| --- | --- |
|  | Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.like("name","Mou%",MatchMode.ANYWHERE));  List<Product> results = crit.list(); |

Above example uses an org.hibernate.criterion.MatchMode object to specify how to match the specified value to the stored data. The MatchMode object (a type-safe enumeration) has four different matches:

ANYWHERE: Anyplace in the string  
END: The end of the string  
EXACT: An exact match  
START: The beginning of the string

**iv) Restrictions.isNull() and Restrictions.isNotNull() Example**

The isNull() and isNotNull() restrictions allow you to do a search for objects that have (or do not have) null property values.

|  |  |
| --- | --- |
|  | Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.isNull("name"));  List<Product> results = crit.list(); |

**v) Restrictions.gt(), Restrictions.ge(), Restrictions.lt() and Restrictions.le() Examples**

Several of the restrictions are useful for doing math comparisons. The greater-than comparison is gt(), the greater-than-or-equal-to comparison is ge(), the less-than comparison is lt(), and the less-than-or-equal-to comparison is le(). We can do a quick retrieval of all products with prices over $25 like this, relying on Java’s type promotions to handle the conversion to Double:

|  |  |
| --- | --- |
|  | Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.gt("price", 25.0));  List<Product> results = crit.list(); |

**vi) Combining Two or More Criteria Examples**

Moving on, we can start to do more complicated queries with the Criteria API. For example, we can combine AND and OR restrictions in logical expressions. When we add more than one constraint to a criteria query, it is interpreted as an AND, like so:

|  |  |
| --- | --- |
|  | Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.lt("price",10.0));  crit.add(Restrictions.ilike("description","mouse", MatchMode.ANYWHERE));  List<Product> results = crit.list(); |

Difference between criteria query and named query in hibernate?

Criteria query is a query based on criteria such as you want to get those employee records only whose salary is less than 50000, you need to specify criteria here.  
To simply use one query many times, we can named this query and reuse it again and again, so we don't need to write same query again and again. It is known as named query.

## Hibernate Concepts - State of Objects

## The three states of objects

An object can have three states: transient, persistent and detached.

When it is just created and has no primary key, the state is called transient.

Car car = new Car();

car.setName(“Porsche”);

When the session is opened and the object is just saved in or retrieved from the database. This state is called persistent. During this state Hibernate manages the object and saves your changes, if you commit them. Below you can see an example. A car is saved and the name is changed afterwards. As the car is in persistent state, the new name will be saved.

Session session = HibernateSessionFactory.currentSession();

tx = session.beginTransaction();

session.save(car);

car.setName(“Peugeot”);

tx.commit();

The following code loads a car by id and changes the name. There is no *session.update* involved. Every object which is loaded by *session.get*, *session.load* or a query is in persistent state. It is stored in the persistence context of the session. When *tx.commit()* is called, Hibernate will flush the persistence context and all not yet written insert, update and delete statements are executed.

Session session = HibernateSessionFactory.currentSession();

tx = session.beginTransaction();

Car car = (Car) session.get(Car.class, 4711);

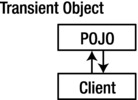
car.setName(“Peugeot”);

tx.commit();

When the session was closed, the state changes to detached. The object is detached from its session

**Transient Object**

Transient objects exist in heap memory. Hibernate does not manage transient objects or persist changes to transient objects.

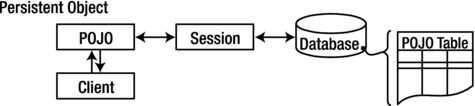


Transient objects are independent of Hibernate

To persist the changes to a transient object, you would have to ask the session to save the transient object to the database, at which point Hibernate assigns the object an identifier and marks the object as being in persistent state.

**Persistent Object**

Persistent objects exist in the database, and Hibernate manages the persistence for persistent objects.



Persistent objects are maintained by Hibernate

If fields or properties change on a persistent object, Hibernate will keep the database representation up to date when the application marks the changes as to be committed.

**Detached Object**

Detached objects have a representation in the database, but changes to the object will not be reflected in the database, and vice-versa. This temporary separation of the object and the database is shown in image below.



Detached objects exist in the database but are not maintained by Hibernate

A detached object can be created by closing the session that it was associated with, or by evicting it from the session with a call to the session’s evict() method.

Native SQL API

a) Auto ResultSet Handing

While using native SQL with hibernate, we need not to map the result set with Entity, instead hibernate does it automatically out of the box for us. To do this, we need to use addEntity(Entity) method on SQL query object.

Hibernate automatically maps the returned values with the object properties based on the configuration done in hbm xml file. For example below query will return the list of Books object.

List result = session.createSQLQuery("select \* from Books ").addEntity(Book.class).list();

**b) Fetching Scalar Values**

If we do not map entity using addEntity() method, query will return the result as scalar values. In such case the result list will be list of Object[] (array of Object)  where each field of object array will be scalar type like String, Integer.

In case we are fetching some of the properties in SQL (not using \* ) then we need to explicitly tell Hibernate about it using addScalar() method like below

session.createSQLQuery("select b.NAME as name from BOOKS b").addScalar("name");

c) Working with Joins

Suppose a entity has a collection of another entity then to load the contained collection at the same time we can use addJoin().

Example:

Let's consider a relationship of Employee and Job. Employee object will have an instance of Job.   Have a look at below mapping.xml

<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-mapping PUBLIC

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

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

<hibernate-mapping>

<class name="com.Employee" table="Employee">

     <id name="id" type="int" column="id">

        <generator class="native" />

     </id>

<property name="name" column="name" type="string" />

    <property name="emailAddress" column="emailAddress" type="string" />

<many-to-one class="com.Job" name="job" column="job\_id" unique="true"/>

</class>

<class table="Job" name="com.Job" >

     <id name="id" type="int" column="id">

        <generator class="native" />

    </id>

<property name="designation" column="designation" type="string"/>

     <property name="salary" column="salary" type="int"/>

     <property name="jobDescription" column="job\_description" type="string"/>

</class>

</hibernate-mapping>

Existing Sample Data in tables

Employee Table –

http://www.wideskills.com/sites/default/files/subjects/Hibernate/Images/17/Image-1.png

Job Table

Case 1- Get All properties of Employee table

Session session = factory.openSession();

SQLQuery query = session.createSQLQuery("select \* from Employee ");

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

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

{

    System.out.println("Details of Employee " + (i+1));

Object[] data = result.get(i);

    System.out.println("Employee ID " + data[0]);

    System.out.println("Employee Name " + data[1]);

    System.out.println("Email Address " + data[2]);

    System.out.println("=====================");

}

 Case 2- Get All properties of Employee table as an Entity

Session session = factory.openSession();

SQLQuery query = session.createSQLQuery("select \* from Employee ");

query.addEntity(Employee.class);

List<Employee> result = query.list();

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

{

    System.out.println("Details of Employee " + (i+1));

Employee emp  = result.get(i);

    System.out.println("Employee ID " + emp.getId());

    System.out.println("Employee Name " + emp.getName());

    System.out.println("Email Address " + emp.getEmailAddress());

    System.out.println("=====================");

}

Result

http://www.wideskills.com/sites/default/files/subjects/Hibernate/Images/17/Image_3.png

Case 3- Get Data from multiple tables

Session session = factory.openSession();

SQLQuery query = session.createSQLQuery("select e.id, e.name, e.emailAddress,"

+ " j.designation,j.salary , j.job\_description  from Employee e  , Job j where e.job\_id= j.id ");

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

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

{

    System.out.println("Details of Employee " + (i+1));

Object[] data = result.get(i);

    System.out.println("Employee ID " + data[0]);

    System.out.println("Employee Name " + data[1]);

    System.out.println("Email Address " + data[2]);

    System.out.println("Designation " + data[3]);

    System.out.println("Salary " + data[4]);

    System.out.println("Job Description " + data[5]);

  System.out.println("=====================");

}

Result

http://www.wideskills.com/sites/default/files/subjects/Hibernate/Images/17/Image_5.png

Case 4- Get Data from multiple tables using Join and as Entity

Returned Result set will be List an array of Object (one for each row ). For each row it will be an array of size 2. One for Employee and one for Job. Setting join is like mapping the alias name with the property name of parent class.

Session session = factory.openSession();

SQLQuery query = session.createSQLQuery("select {emp.\*}, {j.\*} from Employee emp join Job j ON emp.job\_id=j.id");

query.addEntity("emp",Employee.class);

query.addJoin("j","emp.job" );

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

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

{

    System.out.println("Details of Employee " + (i+1));

Object[] data = result.get(i);

Employee emp = (Employee)data[0];

    Job job = (Job) data[1];

System.out.println("Employee ID " + emp.getId());

    System.out.println("Employee Name " + emp.getName());

    System.out.println("Email Address " + emp.getEmailAddress());

System.out.println("Designation " + job.getDesignation());

    System.out.println("Job Description " + job.getJobDescription());

    System.out.println("Salary " + job.getSalary());

System.out.println("=====================");

}

Result

http://www.wideskills.com/sites/default/files/subjects/Hibernate/Images/17/Image_6.png

**Named Queries:**

Both HQL and native SQL can be externalized and can be defined in  mapping file instead of writing it in a code. There are certain advantages of using named queries like

* Syntax of the query is checked at the time of building session factory which helps in knowing about the error earlier.
* The queries become more like configurations and can be managed without changing the code.
* All queries can be kept at a central location.

We need to use getNamedQuery(“Query Name “)  API of session.

HQL Named Query:

HQL named query can be define with <query> tag. Name attribute has to be used to assign a name to the Query and it has to be unique.  Since this is a HQL, queries will written with reference to Java entity . Refer below sample

       <query name="getEmployeeDetails">

                <![CDATA[from Employee]]>

        </query>

Example

Let’s create a new mapping file or add below queries in existing mapping file. Make sure that mapping  file has been added in hibernate.cfg.xml file.

<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-mapping PUBLIC

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

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

<hibernate-mapping package="com.hibernate">

<query name="GET\_ALL\_EMPLOYEES" >

          <![CDATA[from Employee]]>

   </query>

<query name="GET\_EMPLOYEE\_BY\_ID" >

          <![CDATA[from Employee where id = :id]]>

   </query>

</hibernate-mapping>

Create a Test Program to execute the queries

Session session = factory.openSession();

Query query = session.getNamedQuery("GET\_ALL\_EMPLOYEES");

List<Employee> employees = query.list();

for(int i=0;i<employees.size();i++)

{

    System.out.println("Details of Employee " + (i+1));

    Employee emp = employees.get(i);

Job job = emp.getJob();

    System.out.println("Employee ID " + emp.getId());

    System.out.println("Employee Name " + emp.getName());

    System.out.println("Email Address " + emp.getEmailAddress());

System.out.println("Designation " + job.getDesignation());

    System.out.println("Job Description " + job.getJobDescription());

    System.out.println("Salary " + job.getSalary());

System.out.println("=====================");

}

query = session.getNamedQuery("GET\_EMPLOYEE\_BY\_ID");

query.setInteger("id", 1);

Employee emp = (Employee)query.uniqueResult();

Job job = emp.getJob();

System.out.println("Employee ID " + emp.getId());

System.out.println("Employee Name " + emp.getName());

System.out.println("Email Address " + emp.getEmailAddress());

System.out.println("Designation " + job.getDesignation());

System.out.println("Job Description " + job.getJobDescription());

System.out.println("Salary " + job.getSalary());

System.out.println("=====================");

http://www.wideskills.com/sites/default/files/subjects/Hibernate/Images/17/Image_7.png

Native SQL Named Query:

Native SQL named query can be defined with <sql-query> tag. Name attribute has to be used to assign a name to the Query and it has to be unique.  Since this is a native SQL, queries will be written with reference to database table.  Refer below sample

<sql-query name="getEmployeeDetails">

           <![CDATA[select \* from  Employee]]>

    </sql-query>

**Example**

Let’s create a new mapping file or add below queries in an existing mapping file. Make sure that mapping  file has been added in hibernate.cfg.xml file.

<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-mapping PUBLIC

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

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

<hibernate-mapping package="com.hibernate">

<sql-query name="SQL\_GET\_ALL\_EMPLOYEES">

        <![CDATA[select \* from Employee]]>

   </sql-query>

<sql-query name="SQL\_GET\_ALL\_EMPLOYEES\_JOIN">

        <![CDATA[select {emp.\*}, {j.\*} from Employee emp join Job j ON emp.job\_id=j.id ]]>

        <return alias="emp" class="Employee" />

        <return-join alias="j" property="emp.job"></return-join>

    </sql-query>

</hibernate-mapping>

Create a Test Program to execute the queries

Session session = factory.openSession();

Query query = session.getNamedQuery("SQL\_GET\_ALL\_EMPLOYEES\_JOIN");

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

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

{

    System.out.println("Details of Employee " + (i+1));

Object[] data = result.get(i);

Employee emp = (Employee)data[0];

    Job job = (Job) data[1];

System.out.println("Employee ID " + emp.getId());

    System.out.println("Employee Name " + emp.getName());

    System.out.println("Email Address " + emp.getEmailAddress());

System.out.println("Designation " + job.getDesignation());

    System.out.println("Job Description " + job.getJobDescription());

    System.out.println("Salary " + job.getSalary());

System.out.println("=====================");

}

query = session.getNamedQuery("SQL\_GET\_ALL\_EMPLOYEES");

result = query.list();

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

{

    System.out.println("Details of Employee " + (i+1));

Object[] data = result.get(i);

    System.out.println("Employee ID " + data[0]);

    System.out.println("Employee Name " + data[1]);

    System.out.println("Email Address " + data[2]);

    System.out.println("=====================");

}

**Inheritance in Hibernate:**

**The three approaches adopted by Hibernate are**

1. **table-per-class-hierarchy:** Only **one table is created** for all the classes involved in hierarchy. Here we maintain an extra **discriminator field** in table to differentiate between two tables.
2. **table-per-subclass:** **One table for each class is created**. Here, foreign key is maintained between the tables.
3. **table-per-concrete-class:** **One table for each concrete class (subclass) is created but not of super class**. As a special case, the super class can be an abstract or interface. Here, foreign key is not maintained.

**Associations in Hibernate:**

**Managing associations such as one-to-many, many-to-one, many-to-many or one-to-one using hibernate framework is as simple as working with collection properties such as java.util.Set, List, Map.**

**Associations w.r.t domain modeling**

**-----------------------------------------------**

**1) The one-to-many association in Relational modeling is represented as one of the collection property such as java.util.Set, List, Map in domain modeling.**

**2) The foreign key column in relational modeling is represented as object of the associated pojo in the domain modeling.**

**Example:**

**public class Team implements Serializable{**

**private int id; //id property**

**private String name; //common property**

**private Set<Player> players; //collection property**

**public void setPlayers(Set<Player> players){this.players = players;}**

**public Set<Player> getPlayers(){return this.players;}**

**...**

**}**

**public class Player implements Serializable{**

**private int id; //id property**

**private String name; //common property**

**private int age; //common property**

**private Team team; //foreign key property**

**public void setTeam(Team team){this.team = team;}**

**public Team getTeam(){return this.team;}**

**...**

**}**

**Conclusion:**

**1) The primary key column in relation modeling becomes id property in domain modeling.**

**2) The common column in relational modeling becomes common property in domain modeling.**

**3) The one-to-many association in relatinal modleing becomes collection property in domain modeling.**

**4) The foreign key column in relatinal modleing becomes foreign key property in domain modeling.**

**Associations w.r.t hib mapping files**

**------------------------------------**

**#Team.hbm.xml**

**<hibernate-mapping>**

**<class name="Team" table="TEAM">**

**<!--id property -->**

**<id>**

**<!--common property -->**

**<property>**

**<!--collection property -->**

**<set name="players" table="PLAYER" cascade="none | all | save-update | persist | merge | ..." lazy="true | false" inverse="false|true">**

**<!--foreign key column name-->**

**<key column="TEAM\_FK"/>**

**<one-to-many class="Player"/>**

**</set>**

**</class>**

**</hibernate-mapping>**

**<set name="collection property name" [players]**

**table="associated table name" [PLAYER]**

**cascade="none | all | save-update | persist | merge | ..." [none]**

**lazy="true | false" [true]**

**inverse="false | true" [false]>**

**#Player.hbm.xml**

**<hibernate-mapping>**

**<class name="Player" table="PLAYER">**

**<!--id property -->**

**<id>**

**<!--common property -->**

**<property>**

**<property>**

**<!--foreign key property -->**

**<many-to-one name="team" column="TEAM\_FK" class="Team"/>**

**</class>**

**</hibernate-mapping>**

**Conclusion:**

**1) The id property in domain modeling is defined using <id> in mapping file.**

**2) The common property in domain modeling is defined using <property> in mapping file.**

**3) The collection property in domain modeling is defined using <list> or <set> or <map> in mapping file.**

**4) The foreign key property in domain modeling is defined using <many-to-one> in mapping file.**

**Cache Mechanism:**

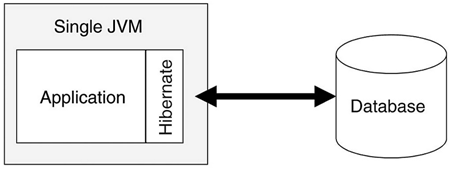
**Caching** is the most preferred technique to solve this problem. **Caching is nothing but some buffer where a record is stored when first time retrieved from the database.** When second time needed the same record, Hibernate does not access the database and instead reads from the **cache**. This type of adjustment decreases the database hits. This is what Cache Hibernate is. Accessing cache is much faster than accessing the database.

**Cache Hibernate implementations**

Hibernate comes with two types of cache mechanism – **First-level cache** and **Second-level cache**.

**1. First-level cache with Session Object**

First-level cache is associated with **Session** object. It is the default cache Hibernate uses (where programmer need not write any extra code).

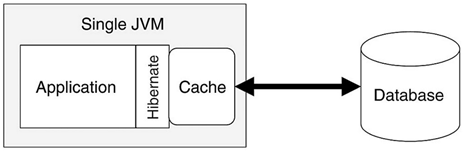
[](http://way2java.com/wp-content/uploads/2013/12/image41.png)

How it works? Hibernate does not process each database hit separately and instead preserves the query in the buffer memory (associated with Session). Infact, it writes one **addBatch()** statement and stores with the **Session** object. When the transaction is committed or session is flushed, Hibernate executes all the queries, stored earlier, with **executeBatch()** statement. This obviously increases the performance.

Session cache is useful on a per-transaction basis. What does it mean? Let us see clearly. As the first level cache is associated with the **Session** object, its scope is limited to one session only. If we fetch a record (better call as persistent object in Hibernate), say with [get()](http://way2java.com/hibernate/hibernate-get-load-delete-and-update-methods/) method, from the database, it is stored with the Session cache. If the same record is fetched again second time, database hit is not made as it gets from the **Session** cache. This you can see in the Console screen in the coming programs. If you fetch the same record with another **Session** object, database hit is made.

**2. Second-level cache with SessionFactory Object**

Second-level cache always associates with the **SessionFactor**y object. While running the transactions, whatever records (or persistent objects) the session object fetches from the database are preserved in **SessionFactory cache (buffer)**. These records are available not only to the current Session object but all Session objects created from SessionFactory object. Any Session object requires the same record again, the database is not hit and instead reads from the **SessionFactory cache**. This we can prove later in the client program.

[](http://way2java.com/wp-content/uploads/2013/12/image42.png)

The entire applications can access the **SessionFactory** cache. It is like **ServletContext** in case of servlets. If you would like the object data should be available to all the threads in the client program, the best choice is second-level cache.

Hibernate comes with four open-source cache implementations to support second-level caching.

1**. EHCache (Easy Hibernate Cache)  
2. OSCache (Open Symphony Cache)  
3. Swarm Cache  
4. JBoss Tree Cache.**

Each style differs in their performance, memory usage and support for different properties supported by servers like clustering etc.

For this program of Cache Hibernate, we use **EHCache**. EHCache comes by default with Hibernate software.