Hibernate

Introduction

- The Hibernate 3.0 core is 68,549 lines of Java code together with 27,948 lines of unit tests, all freely available under the LGPL.

- Hibernate maps the Java classes to the database tables. It also provides the data query and retrieval facilities that significantly reduces the development time.

- It is most useful with object-oriented domain modes and business logic in the Java-based middle-tier.

- Hibernate allows transparent persistence that enables the applications to switch any database.

Features of Hibernate

- Hibernate 3.0 provides three full-featured query facilities: **Hibernate Query Language**, the newly enhanced **Hibernate Criteria Query API**, and enhanced support for queries expressed in the **native SQL** dialect of the database

- Enhanced Criteria query API: with full support for projection/aggregation and subselects

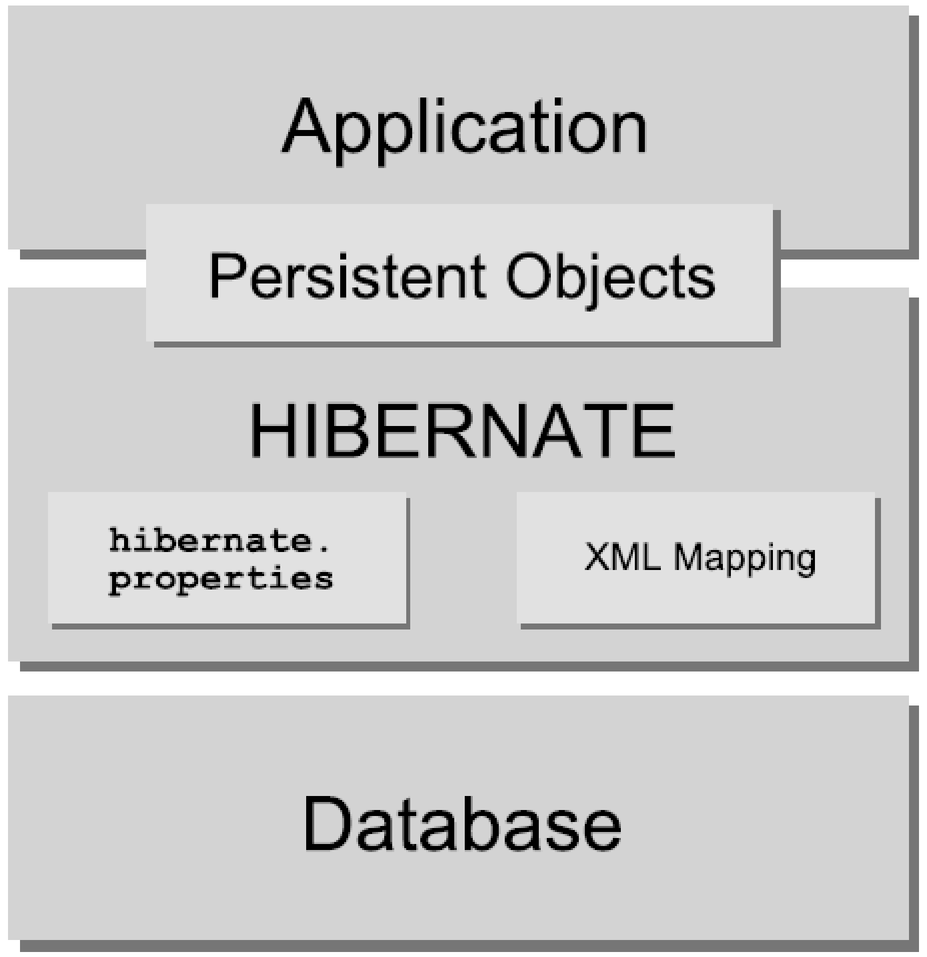
Features

- Eclipse support, including a suite of Eclipse plug-ins for working with Hibernate 3.0
- Hibernate is Free under LGPL: Hibernate can be used to develop/package and distribute the applications for free.
- Hibernate is Scalable: Hibernate is very performant and due to its dual-layer architecture can be used in the clustered environments.
- Less Development Time: Hibernate reduces the development timings as it supports inheritance, polymorphism, composition and the Java Collection framework.

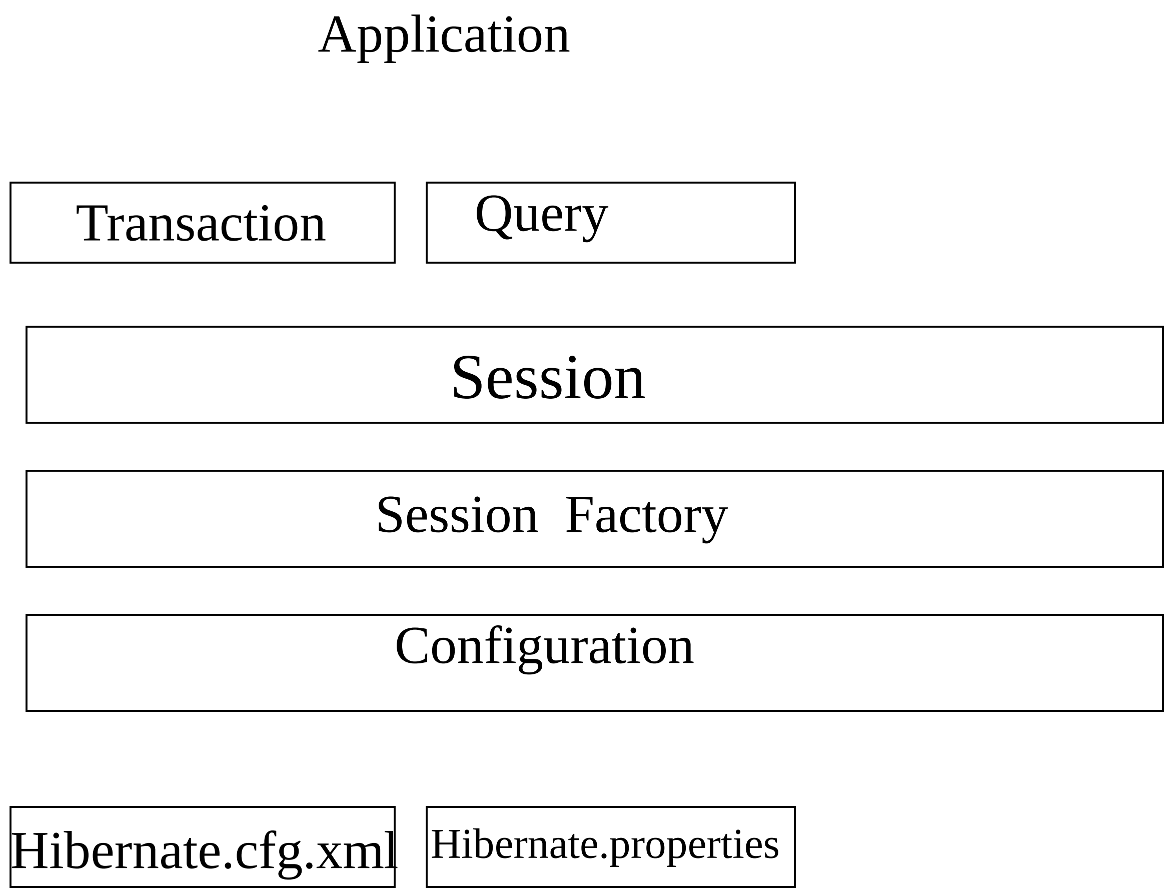
Features

- Automatic Key Generation: Hibernate supports the automatic generation of primary key for your.
- Hibernate XML binding enables data to be represented as XML and POJOs interchangeably.

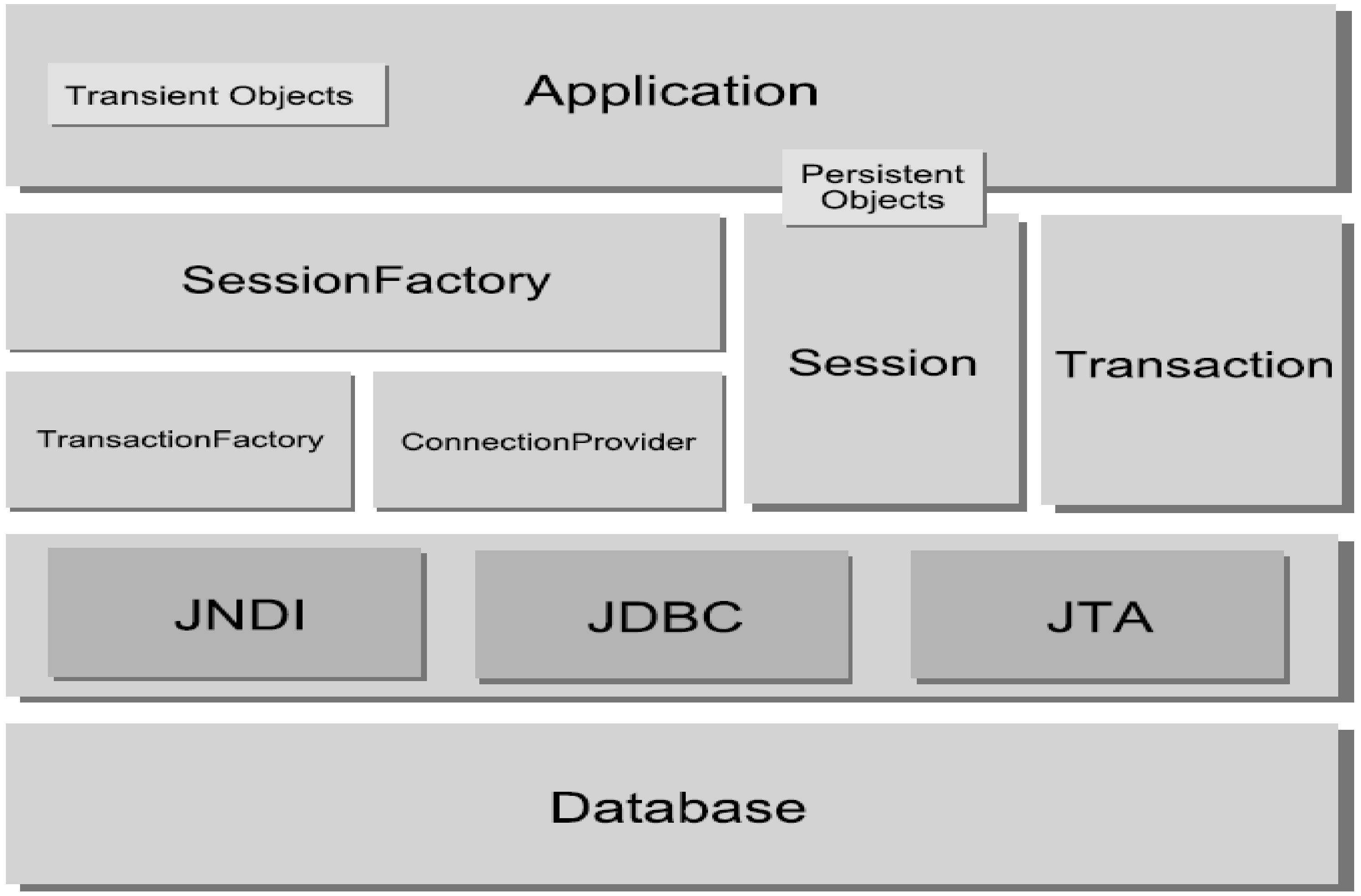
Hibernate Architecture



Hibernate Architecture



Hibernate Architecture



- To use Hibernate, it is required to create Java classes that represents the table in the database and then map the instance variable in the class with the columns in the database. Then Hibernate can be used to perform operations on the database like select, insert, update and delete the records in the table. Hibernate automatically creates the query to perform these operations.

Main Components of Hibernate

- **Connection Management**
  Hibernate Connection management service provide efficient management of the database connections. Database connection is the most expensive part of interacting with the database as it requires a lot of resources of open and close the database connection.

- **Transaction management:**
  Transaction management service provide the ability to the user to execute more than one database statements at a time.

Main Components of Hibernate

- **Object relational mapping:**
  Object relational mapping is technique of mapping the data representation from an object model to a relational data model. This part of the hibernate is used to select, insert, update and delete the records form the underlying table. When we pass an object to a **Session.save()** method, Hibernate reads the state of the variables of that object and executes the necessary query.

- Hibernate is very good tool as far as object relational mapping is concern, but in terms of connection management and transaction management, it is lacking in performance and capabilities. So usually hibernate is being used with other connection management and transaction management tools. For example apache DBCP is used for connection pooling with the Hibernate.

- Hibernate provides a lot of flexibility in use. It is called "**Lite**" architecture when we only uses the object relational mapping component. While in "**Full Cream**" architecture all the three component Object Relational mapping, Connection Management and Transaction Management) are used.

Core Interfaces

- Session Interface :- It is the primary interface used by all Hibernate Applications. It is lightweight and inexpensive to create and destroy. They are not thread safe and be used by only one thread at a time.
- It is also called Persistent Manager, as it is the interface for persistent related operations such as storing and retrieval.
- Session is Something between Connection and Transaction.
- Session is a cache or collection of loaded objects related to single unit of work.

- Session Factory Interface:- The application obtains Session instances from SessionFactory.
- It is not lightweight and intended to be shared among many application threads.
- Single SessionFactory for the whole Application.
- It Caches generated SQL statements and other mapping meatadata that hibernate uses at runtime.
- It also caches data read in one unit of work and may be used another unit of work.

- Configuration Interface:-- It is used to configure and bootstrap Hibernate.
- Application uses Configuration to specify the location of mapping documents and Hibernate specific properties to create SessionFactory.
- It is the first Object used in Hibernate Application Scope.

- Transaction Interface:-- It is Optional API.
- It abstracts application code from the underlying transaction implementation- which might be JDBC Transaction, a JTA UserTransaction.
- It helps to Hibernate Application portable between different kinds execution environments and containers.

- Query and Criteria Interfaces:--Query allows to perform queries against the database and control how the query is executed. Queries are written in HQL or in the native SQL dialect of your database.
- A query instance is used to bind query parameters, limit the number of results and finally to execute the query.
- Criteria Interface is very similar: it allows to create and execute Object Oriented Criteria queries.
- A Query is lightweight and cant be used outside the session that created it.

Types

- A Hibernate Type Object maps a Java type to a Database Column type.
- All Persistent properties of persistent classes have a corresponding Hibernate type.
- It supports all Primitive types and JDK classes like java.util.Currency, Calendar, Serializable.

Creating SessionFactory

- First Create an single instance of Configuration during application initialization and use it to set the location of the mapping files.
- The Configuration instance is used to create Session Factory.
- SessionFactory sessionFactory=new Configuration().configure().buildSessionFactory();

Configuring Hibernate

- **Configuring Hibernate**
  In this application Hibernate provided connection pooling and transaction management is used for simplicity. Hibernate uses the hibernate.cfg.xml to create the connection pool and setup required environment.

```
<hibernate-configuration>
<session-factory> <property
name="hibernate.connection.driver_class">oracle.jdbc.driver.Oracl
eDriver</property>
<property
name="hibernate.connection.url">jdbc:oracle:thin:@localhost:15
21:orcl</property>
    <property
name="hibernate.connection.username">scott</property>
    <property
name="hibernate.connection.password">tiger</property>
    <property
name="hibernate.connection.pool_size">10</property>
```

```
<property name="show_sql">true</property>
      <property
  name="dialect">org.hibernate.dialect.Oracle9Dialect
   </property>

      <!-- Mapping files -->
      <mapping resource="contact.hbm.xml"/>
  </session-factory></hibernate-configuration>
```

- With the use of the Hibernate (Object/Relational Mapping and Transparent Object Persistence for Java and SQL Databases), we can use the following databases dialect type property

- DB2 - org.hibernate.dialect.DB2Dialect
- HypersonicSQL - org.hibernate.dialect.HSQLDialect
- Informix - org.hibernate.dialect.InformixDialect
- Ingres - org.hibernate.dialect.IngresDialect
- Interbase - org.hibernate.dialect.InterbaseDialect
- Pointbase - org.hibernate.dialect.PointbaseDialect
- PostgreSQL - org.hibernate.dialect.PostgreSQLDialect
- Microsoft SQL Server-org.hibernate.dialect.SQLServerDialect
- MySQL - org.hibernate.dialect.MySQLDialect
- Oracle (any version) - org.hibernate.dialect.OracleDialect
- Oracle 9 - org.hibernate.dialect.Oracle9Dialect
- Progress - org.hibernate.dialect.ProgressDialect
- Sybase - org.hibernate.dialect.SybaseDialect
- Sybase Anywhere - org.hibernate.dialect.SybaseAnywhereDialect

The <mapping resource="**contact.hbm.xml**"/> property is the mapping for our contact table.

- **Writing First Persistence Class**
  Hibernate uses the Plain Old Java Objects (POJOs) classes to map to the database table. We can configure the variables to map to the database column. Here is the code for Contact.java

```
package dev;
   //  Java Class to map to the datbase Contact Table
   public class Contact {
     private String firstName;
     private String lastName;
     private String email;

  private long id;
    public String getEmail()

   {    return email;  }

public String getFirstName()

   { return firstName; }

public String getLastName()

{    return lastName; }} }
```

```
public void setEmail(String string) {email = string;}

public void setFirstName(String string)
{  firstName = string;  }
     public void setLastName(String string)
{    lastName = string; }
     public long getId()
{    return id;  }
     public void setId(long l)
{   id = l;  }}
```

Mapping the Contact Object to the Database Contact table

The file contact.hbm.xml is used to map Contact Object to the Contact table in the database. Here is the code for contact.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=“dev.Contact" table="CONTACT">
     <id name="id" type="long" column="ID" >
     <generator class="assigned"/> </id>
  <property name="firstName">
     <column name="FIRSTNAME" />
    </property>
    <property name="lastName">
      <column name="LASTNAME"/>
    </property>
    <property name="email">
      <column name="EMAIL"/>
    </property>
   </class></hibernate-mapping>
  ```

Developing Code to Test Hibernate example

- Now we are ready to write a program to insert the data into database. We should first understand about the Hibernate's Session. Hibernate Session is the main runtime interface between a Java application and Hibernate. First we are required to get the Hibernate Session.SessionFactory allows application to create the Hibernate Sesssion by reading the configuration from hibernate.cfg.xml file. Then the save method on session object is used to save the contact information to the database: **session.save(contact)**

```
  package dev;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class FirstExample {
  public static void main(String[] args) {
    Session session = null;   try{
      SessionFactory sessionFactory =

          new Configuration().configure().buildSessionFactory();
       session =sessionFactory.openSession();
         System.out.println("Inserting Record");
Contact contact = new Contact();
        contact.setId(3);

    contact.setFirstName("Deepak");
        contact.setLastName("Kumar");
        contact.setEmail("deepak_38@yahoo.com");
        session.save(contact);

        System.out.println("Done");    }

     catch(Exception e){
       System.out.println(e.getMessage()); }

       finally{
       session.flush();

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

Understanding Hibernate O/R Mapping

1. **<hibernate-mapping> element**
   The first or root element of hibernate mapping document is <hibernate-mapping> element. Between the <**hibernate-mapping>** tag class element(s) are present.

2. **<class> element**
   The <Class> element maps the class object with corresponding entity in the database. It also tells what table in the database has to access and what column in that table it should use. Within one <hibernate-mapping> element, several <class> mappings are possible .

Understanding Hibernate O/R Mapping

3. <id> element

The <id> element in unique identifier to identify and object. In fact <id> element map with the primary key of the table. In our code :
<id name="id" type="long" column="ID" >
primary key maps to the **ID** field of the table **CONTACT**. The attributes of the id element are:

- **name:** The property name used by the persistent class.
- **column:** The column used to store the primary key value.
- **type:** The Java data type used.

Understanding Hibernate O/R Mapping

4. <generator> element

The **<generator>** method is used to generate the primary key for the new record. Here is some of the commonly used generators :

* **Increment** - This is used to generate primary keys of type long, short or int that are unique only. It should not be used in the clustered deployment environment.

* **Sequence** - Hibernate can also use the sequences to generate the primary key. It can be used with DB2, PostgreSQL, Oracle, SAP DB databases.

* **Assigned** - Assigned method is used when application code generates the primary key.

Understanding Hibernate O/R Mapping

5. <property> element

The property elements define standard Java attributes and their mapping into database schema. The property element supports the column child element to specify additional properties, such as the index name on a column or a specific column type.

The <generator> element in Detail

- This is the optional element under <id> element. The <generator> element is used to specify the class name to be used to generate the primary key for new record while saving a new record. Here is the example of generator element from our first application:
  <generator class="assigned"/>
  In this case <generator> element do not generate the primary key and it is required to set the primary key value before calling save() method.

```
package dev;
   public class Book {

 private long lngBookId;
     private String strBookName;

     public long getLngBookId()
       {    return lngBookId;  }

     public void setLngBookId(long lngBookId)
     {    this.lngBookId = lngBookId;  }

     public String getStrBookName() {return strBookName;  }

    public void setStrBookName(String strBookName) {
      this.strBookName = strBookName;
      }
}
```

Adding Mapping entries to contact.hbm.xml

-
  ```
  <class name=“dev.Book" table="book">
      <id name="lngBookId" type="long" column="id" >
        <generator class="increment"/>
      </id>

     <property name="strBookName">
         <column name="bookname" />
      </property>
  </class>
  ```

Client Program to Test

- **package** dev;
- **import** org.hibernate.*;
- **import** org.hibernate.cfg.Configuration;
  public class IdIncrementExample {
   public static void main(String[] args) {
    Session session = **null**;
  try{ SessionFactory sessionFactory =

   new Configuration().configure().buildSessionFactory();
   session =sessionFactory.openSession();
   org.hibernate.Transaction tx = session.beginTransaction();

- System.out.println("Inserting Book object into database..");
  Book book = **new** Book();
  book.setStrBookName("Hibernate Tutorial");
  session.save(book);
    tx.commit();
- session.flush();

  session.close();
  }**catch**(Exception e){

   System.out.println(e.getMessage()); }

finally{   }

- }

}

Hibernate Query Language

- Hibernate Query Language or HQL for short is extremely powerful query language. HQL is much like SQL and are case-insensitive, except for the names of the Java Classes and properties. Hibernate Query Language is used to execute queries against database. Hibernate automatically generates the sql query and execute it against underlying database if HQL is used in the application. HQL is based on the relational object models and makes the SQL object oriented. Hibernate Query Language uses Classes and properties instead of tables and columns. Hibernate Query Language is extremely powerful and it supports Polymorphism, Associations, Much less verbose than SQL.

- There are other options that can be used while using Hibernate. These are **Query By Criteria (QBC)** and **Query BY Example (QBE)** using Criteria API and the **Native SQL** queries.

Why To Use HQL?

- **Full support for relational operations:** HQL allows representing SQL queries in the form of objects. Hibernate Query Language uses Classes and properties instead of tables and columns.
- **Return result as Object:** The HQL queries return the query result(s) in the form of object(s), which is easy to use. This elemenates the need of creating the object and populate the data from result set.
- **Polymorphic Queries:** HQL fully supports **polymorphic queries**. Polymorphic queries results the query results along with all the child objects if any.
- **Easy to Learn:** Hibernate Queries are easy to learn and it can be easily implemented in the applications.
- **Database independent:** Queries written in HQL are database independent (If database supports the underlying feature).

Why to use HQL?

- **Support for Advance features:** HQL contains many advance features such as pagination, fetch join with dynamic profiling, Inner/outer/full joins, Cartesian products. It also supports Projection, Aggregation (max, avg) and grouping, Ordering, Sub queries and SQL function calls.
- **Understanding HQL Syntax**
  Any Hibernate Query Language may consist of following elements:-
  Clauses ,Aggregate functions ,Subqueries
- **Clauses in the HQL are:-** from,select,where,order by,group by .
- **Aggregate functions are:**
- avg(...), sum(...), min(...), max(...)
- count(*)
- count(...), count(distinct ...), count(all...)
- **Subqueries**
  Subqueries are nothing but its a query within another query.
  Hibernate supports Subqueries if the underlying database supports it.

- /*Table structure for table `insurance` */
- CREATE TABLE insurance (ID int(11) NOT NULL default 0, insurance_name varchar (50) default NULL,
- invested_amount int (11) default NULL,
- investement_date datetime default NULL, PRIMARY KEY (ID))

- insert into `insurance` values (1,'Car Insurance',1000,'2005-01-05 00:00:00');
- insert into `insurance` values (2,'Life Insurance',100,'2005-10-01 00:00:00');
- insert into `insurance` values (3,'Life Insurance',500,'2005-10-15 00:00:00');
- insert into `insurance` values (4,'Car Insurance',2500,'2005-01-01 00:00:00');
- insert into `insurance` values (5,'Dental Insurance',500,'2004-01-01 00:00:00');
- insert into `insurance` values (6,'Life Insurance',900,'2003-01-01 00:00:00');
- insert into `insurance` values (7,'Travel Insurance',2000,'2005-02-02 00:00:00'); insert into `insurance` values (8,'Travel Insurance',600,'2005-03-03 00:00:00');
- insert into `insurance` values (9,'Medical Insurance',700,'2005-04-04 00:00:00'); insert into `insurance` values (10,'Medical Insurance',900,'2005-03-03 00:00:00'); insert into `insurance` values (11,'Home Insurance',800,'2005-02-02 00:00:00');
- insert into `insurance` values (12,'Home Insurance',750,'2004-09-09 00:00:00');
- insert into `insurance` values (13,'Motorcycle Insurance',900,'2004-06-06 00:00:00'); insert into `insurance` values (14,'Motorcycle Insurance',780,'2005-03-03 00:00:00');

```
package dev;
 import java.util.Date;
   public class Insurance {
     private long lngInsuranceId;
     private String insuranceName;
     private int investementAmount;
     private Date investementDate;


     public String getInsuranceName() {
       return insuranceName;
     }


public void setInsuranceName(String insuranceName) {
   this.insuranceName = insuranceName;  }

     public int getInvestementAmount() {
       return investementAmount;}
public void setInvestementAmount(int investementAmount) {
  this.investementAmount = investementAmount; }


    public Date getInvestementDate() {
      return investementDate;

}


    public void setInvestementDate(Date investementDate) {

            this.investementDate = investementDate;
    }
 public long getLngInsuranceId() {
      return lngInsuranceId;

}
    public void setLngInsuranceId(long lngInsuranceId) {

 this.lngInsuranceId = lngInsuranceId; }

}
```

Adding mappings into contact.hbm.xml file

```
<class name=“dev.Insurance" table="insurance">
<id name="lngInsuranceId" type="long" column="ID" >
 <generator class="increment"/>
</id>
<property name="insuranceName">
 <column name="insurance_name" /> </property>
 <property name="investementAmount">
<column name="invested_amount" />
</property>
 <property name="investementDate">
<column name="investement_date" /> </property>
 </class>
```

Client Code to Test

```
 package dev.hibernate;
import org.hibernate.Session;
import org.hibernate.*;
import org.hibernate.cfg.*;
 import java.util.*;
public class SelectHQLExample {

  public static void main(String[] args) {
  Session session = null;
try{
    SessionFactory sessionFactory =
new Configuration().configure().buildSessionFactory();
     session =sessionFactory.openSession();
 //Using from Clause
      String SQL_QUERY ="from Insurance insurance";
      Query query = session.createQuery(SQL_QUERY);
      for(Iterator it=query.iterate();it.hasNext();){
        Insurance insurance=(Insurance)it.next();
        System.out.println("ID: " + insurance.getLngInsuranceId());
   System.out.println("First Name: " + insurance.getInsuranceName());
      }

        session.close();
   }

  catch(Exception e){
     System.out.println(e.getMessage());
   }

finally{   }

}

}
```

Hibernate Select Clause

- The select clause picks up objects and properties to return in the query result set. Here is the query:
- Select insurance.lngInsuranceId, insurance.insuranceName, insurance.investementAmount, insurance.investementDate from Insurance insurance
- which selects all the rows (**insurance.lngInsuranceId, insurance.insuranceName, insurance.investementAmount, insurance.investementDate**) from Insurance table.

```
package dev;
   import org.hibernate.Session;

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

    import java.util.*;
   public class SelectClauseExample {


     public static void main(String[] args) {
     Session session = null;

    try{
     SessionFactory sessionFactory =

   new Configuration().configure().buildSessionFactory();
    session =sessionFactory.openSession();

 String SQL_QUERY ="Select insurance.lngInsuranceId,

   insurance.insuranceName," +

  "insurance.investementAmount,insurance.investementDate from Insur
   ance insurance";
Query query = session.createQuery(SQL_QUERY);
   for(Iterator it=query.iterate();it.hasNext();)
      {
      Object[] row = (Object[]) it.next();
      System.out.println("ID: " + row[0]);
      System.out.println("Name: " + row[1]);
      System.out.println("Amount: " + row[2]);
     }
      session.close();

 }

    catch(Exception e)

 {
    System.out.println(e.getMessage());
  }

    finally { }
  }
 }
```

HQL Where Clause Example

- Where Clause is used to limit the results returned from database. It can be used with aliases and if the aliases are not present in the Query, the properties can be referred by name. For example:

- from Insurance where lngInsuranceId='1‘

- Where Clause can be used with or without Select Clause.

```
package dev;
import org.hibernate.Session;
import org.hibernate.*;
import org.hibernate.cfg.*;
import java.util.*;
public class WhereClauseExample {
    public static void main(String[] args) {
    Session session = null;
    try{
      SessionFactory sessionFactory =
  new Configuration().configure().buildSessionFactory();
      session =sessionFactory.openSession();
       System.out.println("Query using Hibernate Query Language");
  //Query using Hibernate Query Language
       String SQL_QUERY =" from Insurance as insurance where
             insurance.lngInsuranceId='1'";
       Query query = session.createQuery(SQL_QUERY);
```

- **for**(Iterator it=query.iterate();it.hasNext();){

```
   Insurance insurance=(Insurance)it.next();
   System.out.println("ID: " + insurance.getLngInsuranceId());
   System.out.println("Name: " + insurance.getInsuranceName());
 }
 System.out.println("******************************");
 System.out.println("Where Clause With Select Clause");
//Where Clause With Select Clause
 SQL_QUERY ="Select insurance.lngInsuranceId,
     insurance.insuranceName," +
 "insurance.investementAmount,insurance.investementDate from
     Insurance insurance "+
" where insurance.lngInsuranceId='1'";
 query = session.createQuery(SQL_QUERY);
```

```
for(Iterator it=query.iterate();it.hasNext();){
        Object[] row = (Object[]) it.next();
        System.out.println("ID: " + row[0]);
        System.out.println("Name: " + row[1]);

      }
      System.out.println("*******************************");

        session.close();
    }catch(Exception e){
      System.out.println(e.getMessage());
    }finally{
      }
    }
  }
```