Unit-6 Hibernate

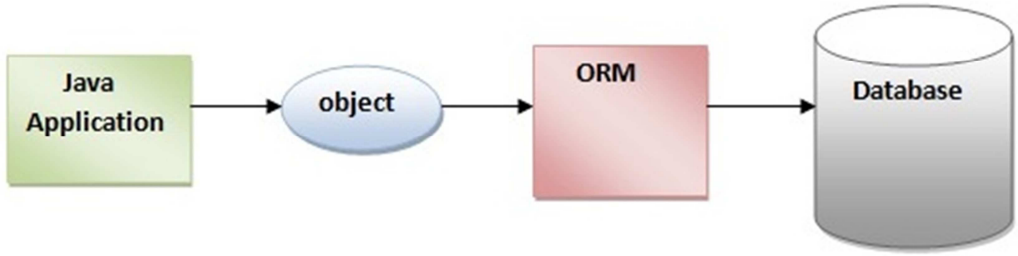
GTU Syllabus:

Hibernate 4.0
Overview of Hibernate, Hibernate Architecture, Hibernate Mapping Types, Hibernate O/R Mapping, Hibernate Annotation, Hibernate Query Language

Overview of Hibernate:

Hibernate framework simplifies the development of java application to interact with the database. Hibernate is an open source, lightweight, ORM (Object Relational Mapping) tool.

An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.



Hibernate is an **O**bject-**R**elational **M**apping (ORM) solution for JAVA. It is an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieves the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.

Advantages of Hibernate Framework

There are many advantages of Hibernate Framework. They are as follows:

1) Opensource and Lightweight: Hibernate framework is opensource under the LGPL license and lightweight.

2) Fast performance: The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled bydefault.

3) Database Independent query: HQL (Hibernate Query Language) is the object-oriented version of SQL. It generates the database independent queries. So you don't need to write database specific queries. Before Hibernate, If database is changed for the project, we need to change the SQL query as well that leads to the maintenance problem.

4) Automatic table creation: Hibernate framework provides the facility to create the tables of the database automatically. So there is no need to create tables in the database manually.

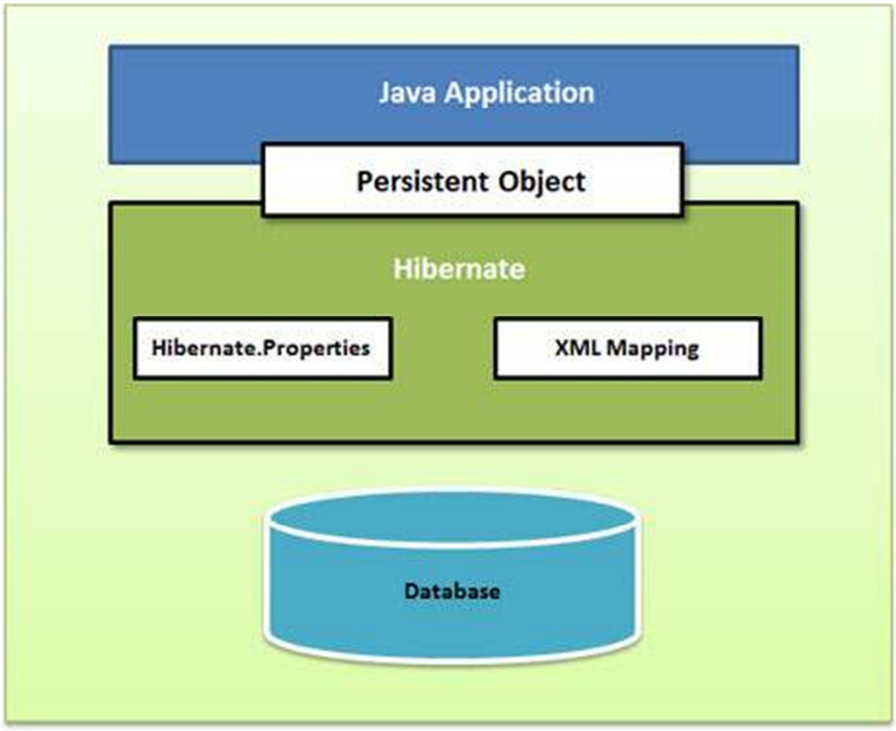
5) Simplifies complex join: To fetch data form multiple tables is easy in hibernate framework.

6) Provides query statistics and database status: Hibernate supports Query cache and provide statistics about query and database status.

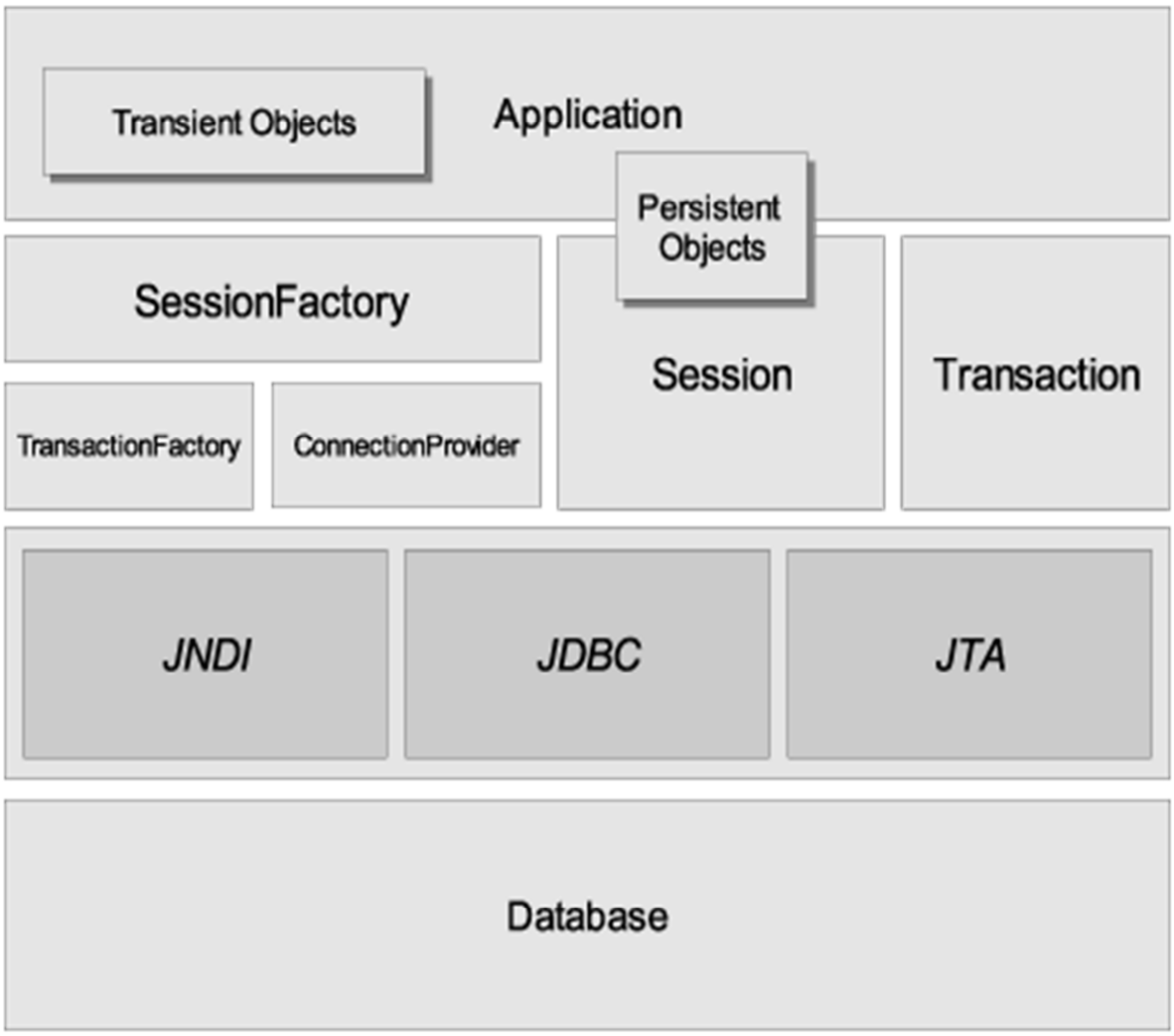
Hibernate Architecture:

Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

Following is a very high level view of the Hibernate Application Architecture.



Following is a detailed view of the Hibernate Application Architecture with its important core classes.



- In order to persist data to a database, Hibernate create an instance of entity class (Java class mapped with database table).
- This object is called Transient object as they are not yet associated with the session or not yet persisted to a database.
- To persist the object to database, the instance of **SessionFactory** interface is created.

SessionFactory is a singleton instance which implements Factory design pattern.SessionFactory loads hibernate.cfg.xml file and with the help of **TransactionFactory** and **ConnectionProvider** implements all the configuration settings on a database.Each database connection in Hibernate is created by creating an interface. Session represents a single connection with database.
Session objects are created from **SessionFactory** object.

- Each transaction represents a single atomic unit of work. One Session can span through multiple transactions.

1. ***SessionFactory (org.hibernate.SessionFactory)***
   - A thread-safe, immutable cache of compiled mappings for a factory for org.hibernate.Session instances.
   - A client of org.hibernate.connection.ConnectionProvider.
   - Optionally maintains a second level cache of data that is transactions at a process or cluster level.

2. ***Session (org.hibernate.Session)***

   - A single-threaded, short-lived object representing a conversation between the application and the persistent store.
   - Wraps a JDBC java.sql.Connection.
   - Factory for org.hibernate.Transaction.
   - Maintains a first level cache of persistent the application's persistent objects and collections; this cache is used when navigating the object graph or looking up objects by identifier.

3. ***Persistent objects and collections***
   - Short-lived, single threaded objects containing persistent function.
   - These can be ordinary JavaBeans/POJOs.
   - They are associated with exactly one org.hibernate.Session.

   state and business
   - Once the org.hibernate.Session is closed, they will be detached and free to use in any application layer.

4. ***Transient and detached objects and collections***
   - Instances of persistent classes that are not currently org.hibernate.Session. associated with a session.
   - They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed org.hibernate.Session.

5. ***Transaction (org.hibernate.Transaction)***
   - A single-threaded, short-lived object used by the application to specify atomic units of work.
   - It abstracts the application from the underlying JDBC, JTA or CORBA transaction.

6. *ConnectionProvider (org.hibernate.connection.ConnectionProvider)*
   - A factory for, and pool of, JDBC connections.
   - It abstracts the application from underlying javax.sql.DataSource or java.sql.DriverManager.
7. *TransactionFactory (org.hibernate.TransactionFactory)*
   - A factory for org.hibernate.Transaction instances. It is not exposed to the application, but it can be extended and/or implemented by the developer.

Hibernate Interfaces:

Package: org.hibernate.*;

org.hibernate.cfg.*;

The five core interfaces are used in just about every Hibernate application. Using these interfaces, you can store and retrieve persistent objects and control transactions.

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

1.Configuration:

The Configuration object is used to configure hibernate. The application uses a Configuration instance to specify the location of mapping documents and Hibernate-specific properties and then create the SessionFactory.

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

2.Transaction:

A transaction represents a unit of work. Application uses transactions to do some operations on DB. Within one transaction you can do several operations and can commit transaction once after successfully completed all operations. The advantage here is you can rollback all previous operations if one operation is fail in your operation batch. The Transaction does not get committed when session gets flushed. The Transaction interface is an optional API. Hibernate applications may choose not to use this interface, instead managing transactions in their own infrastructure code. A

Transaction abstracts application code from the underlying transaction implementation-which might be a JDBC transaction, a JTA UserTransaction, or even a Common Object Request Broker Architecture (CORBA) transaction-allowing the application to control transaction boundaries via a consistent API. This helps to keep Hibernate applications portable between different kinds of execution environments and containers.

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

3.SessionFactory:

The application obtains Session instances from a SessionFactory. SessionFactory instances are not lightweight and typically one instance is created for the whole application. If the application accesses multiple databases, it needs one per database. SessionFactory can hold an optional (second-level) cache of data that is reusable between transactions at a process, or cluster, level.

```
SessionFactory sessionFactory =c.buildSessionFactory();
```

4.Session:

The Session is a persistence manager that manages operation like storing and retrieving objects. Instances of Session are inexpensive to create and destroy. They are not thread safe.

Session holds a mandatory first-level cache of persistent objects that are used when navigating the object graph or looking up objects by identifier.

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

5.Query and Criteria interfaces:

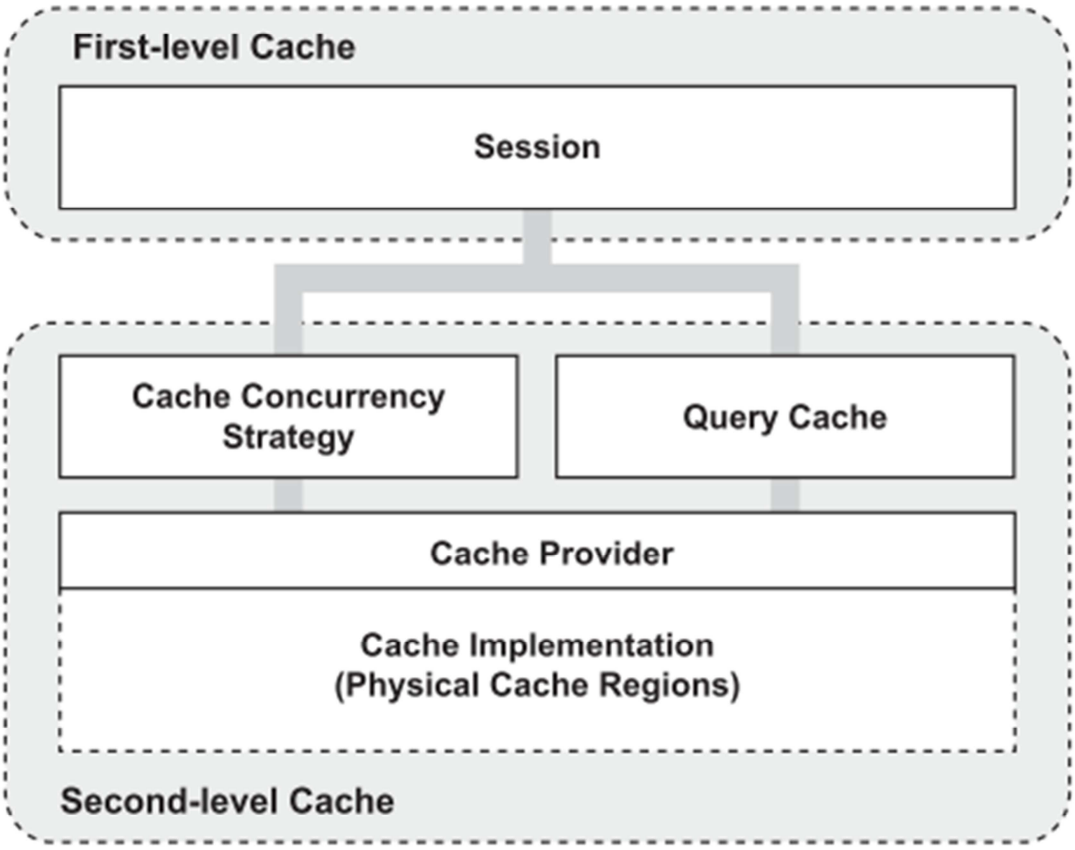
The Query interface allows you to perform queries against the database and control how the query is executed. Queries are written in HQL or in the native SQL dialect of your database. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

The Criteria interface is very similar; it allows you to create and execute object-oriented criteria queries.

To help make application code less verbose, Hibernate provides some shortcut methods on the Session interface that let you invoke a query in one line of code. We won't use these shortcuts; instead, we'll always use the Query interface.

A Query instance is lightweight and can't be used outside the Session that created it.

Hibernate Cache Architecture:



Hibernate's two-level cache architecture

Caching is a mechanism to enhance the performance of a system. It is a buffer memorythat lies between the application and the database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.

First-level Cache

The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.

If you issue multiple updates to an object, Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.

Second-level Cache

Second level cache is an optional cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second level cache can be configured on a per-class and per-collection basis and mainly responsible for caching objects across sessions.

Any third-party cache can be used with Hibernate. An **org.hibernate.cache.CacheProvider** interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation.

Query-level Cache

Hibernate also implements a cache for query resultsets that integrates closely with the second-level cache.

This is an optional feature and requires two additional physical cache regions that hold the cached query results and the timestamps when a table was last updated. This is only useful for queries that are run frequently with the same parameters.

The Second Level Cache

Hibernate uses first-level cache by default and you have nothing to do to use first-level cache. Let's go straight to the optional second-level cache. Not all classes benefit from caching, so it's important to be able to disable the second-level cache.

The Hibernate second-level cache is set up in two steps. First, you have to decide which concurrency strategy to use. After that, you configure cache expiration and physical cache attributes using the cache provider.

Concurrency Strategies

A concurrency strategy is a mediator, which is responsible for storing items of data in the cache and retrieving them from the cache. If you are going to enable a second-level cache, you will have to decide, for each persistent class and collection, which cache concurrency strategy to use.

- **Transactional –** Use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.
- **Read-write –** Again use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.
- **Nonstrict-read-write –** This strategy makes no guarantee of consistency between the cache and the database. Use this strategy if data hardly ever changes and a small likelihood of stale data is not of critical concern.

- **Read-only** – A concurrency strategy suitable for data, which never changes. Use it for reference data only.

Cache Provider

Your next step after considering the concurrency strategies, you will use your cache candidate classes to pick a cache provider. Hibernate forces you to choose a single cache provider for the whole application.

Sr.No.	Cache Name & Description
1	**EHCache** It can cache in memory or on disk and clustered caching and it supports the optional Hibernate query result cache.
2	**OSCache** Supports caching to memory and disk in a single JVM with a rich set of expiration policies and query cache support.
3	**warmCache** A cluster cache based on JGroups. It uses clustered invalidation, but doesn't support the Hibernate query cache.
4	**JBoss Cache** A fully transactional replicated clustered cache also based on the JGroups multicast library. It supports replication or invalidation, synchronous or asynchronous communication, and optimistic and pessimistic locking. The Hibernate query cache is supported.

The Query-level Cache

To use the query cache, you must first activate it using the **hibernate.cache.use_query_cache="true"** property in the configuration file. By setting this property to true, you make Hibernate create the necessary caches in memory to hold the query and identifier sets.

Hibernate Query Language (HQL):

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

Advantage of HQL

- There are many advantages of HQL. They are as follows:
- database independent
- supports polymorphic queries
- easy to learn for Java Programmer

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries which in turns perform action on database.

Although you can use SQL statements directly with Hibernate using Native SQL but I would recommend to use HQL whenever possible to avoid database portability hassles, and to take advantage of Hibernate's SQL generation and caching strategies.

Keywords like SELECT , FROM and WHERE etc. are not case sensitive but properties like table and column names are case sensitive in HQL.

FROM Clause

You will use **FROM** clause if you want to load a complete persistent objects into memory. Following is the simple syntax of using FROM clause:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

If you need to fully qualify a class name in HQL, just specify the package and class name as follows:

```
String hql = "FROM com.Employee";
```

```
Query query = session.createQuery(hql);
List results = query.list();
```

AS Clause

The **AS** clause can be used to assign aliases to the classes in your HQL queries, specially when you have long queries. For instance, our previous simple example would be the following:

```
String hql = "FROM Employee AS E";
Query query = session.createQuery(hql);
List results = query.list();
```

The **AS** keyword is optional and you can also specify the alias directly after the class name, as follows:

```
String hql = "FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

SELECT Clause

The **SELECT** clause provides more control over the result set than the from clause. If you want to obtain few properties of objects instead of the complete object, use the SELECT clause. Following is the simple syntax of using SELECT clause to get just first_name field of the Employee object:

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

It is notable here that **Employee.firstName** is a property of Employee object rather than a field of the EMPLOYEE table.

WHERE Clause

If you want to narrow the specific objects that are returned from storage, you use the WHERE clause. Following is the simple syntax of using WHERE clause:

```
String hql = "FROM Employee E WHERE E.id = 10";
```

```
Query query = session.createQuery(hql);
List results = query.list();
```

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). Following is the simple syntax of using ORDER BY clause:

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

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 as follows:

```
String hql = "FROM Employee E WHERE E.id > 10 " +
        "ORDER BY E.firstName DESC, E.salary DESC ";
Query query = session.createQuery(hql);
List results = query.list();
```

GROUP BY Clause

This clause lets Hibernate pull information from the database and group it based on a value of an attribute and, typically, use the result to include an aggregate value. Following is the simple syntax of using GROUP BY clause:

```
String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " +
        "GROUP BY E.firstName";
Query query = session.createQuery(hql);
List results = query.list();
```

Using Named Paramters

Hibernate supports named parameters in its HQL queries. This makes writing HQL queries that accept input from the user easy and you do not have to defend against SQL injection attacks. Following is the simple syntax of using named parameters:

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
```

```
Query query = session.createQuery(hql);
query.setParameter("employee_id",10);
List results = query.list();
```

UPDATE Clause

Bulk updates are new to HQL with Hibernate 3, and deletes work differently in Hibernate 3 than they did in Hibernate 2. The Query interface now contains a method called executeUpdate() for executing HQL UPDATE or DELETE statements.

The **UPDATE** clause can be used to update one or more properties of an one or more objects. Following is the simple syntax of using UPDATE clause:

```
String hql = "UPDATE Employee set salary = :salary "  +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

DELETE Clause

The **DELETE** clause can be used to delete one or more objects. Following is the simple syntax of using DELETE clause:

```
String hql = "DELETE FROM Employee "  +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

INSERT Clause

HQL supports **INSERT INTO** clause only where records can be inserted from one object to another object. Following is the simple syntax of using INSERT INTO clause:

```
String hql = "INSERT INTO Employee(firstName, lastName, salary)"  +
             "SELECT firstName, lastName, salary FROM old_employee";
Query query = session.createQuery(hql);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

Aggregate Methods

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL and following is the list of the available functions:

S.N.	Functions	Description
1	avg(property name)	The average of a property's value
2	count(property name or *)	The number of times a property occurs in the results
3	max(property name)	The maximum value of the property values
4	min(property name)	The minimum value of the property values
5	sum(property name)	The sum total of the property values

The **distinct** keyword only counts the unique values in the row set. The following query will return only unique count:

```
String hql = "SELECT count(distinct E.firstName) FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

Pagination using Query

There are two methods of the Query interface for pagination.

S.N.	Method & Description
1	**Query setFirstResult(int startPosition)** This method takes an integer that represents the first row in your result set, starting with row 0.
2	**Query setMaxResults(int maxResult)** This method tells Hibernate to retrieve a fixed number **maxResults** of objects.

Using above two methods together, we can construct a paging component in our web or Swing application. Following is the example which you can extend to fetch 10 rows at a time:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
query.setFirstResult(1);
query.setMaxResults(10);
List results = query.list();
```

Differences between SQL and HQL:

SQL is based on a relational database model whereas HQL is a combination of object-oriented programming with relational database concepts.

SQL manipulates data stored in tables and modifies its rows and columns. HQL is concerned about objects and its properties.

SQL is concerned about the relationship that exists between two tables while HQL considers the relation between two objects.

Summary:

1. HQL is similar to SQL and is also case insensitive.

2. HQL and SQL both fire queries in a database. In the case of HQL, the queries are in the form of objects that are translated to SQL queries in the target database.

3. SQL works with tables and columns to manipulate the data stored in it.

4. HQL works with classes and their properties to finally be mapped to a table structure in a database.

5. HQL supports concepts like polymorphism, inheritance, association, etc. It is a powerful and easy-to-learn language that makes SQL object oriented.

6. SQL lets you modify the data through insert, update, and delete queries. You can add tables, procedures, or views to your database. The permissions on these added objects can be changed.

Hibernate Mapping Types:

When you prepare a Hibernate mapping document, you find that you map the Java data types into RDBMS data types. The **types** declared and used in the mapping files are not Java data types; they are not SQL database types either. These types are called **Hibernate mapping types**, which can translate from Java to SQL data types and vice versa.

This chapter lists down all the basic, date and time, large object, and various other builtin mapping types.

Primitive Types

Mapping type	Java type	ANSI SQL Type
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte or java.lang.Byte	TINYINT
boolean	boolean or java.lang.Boolean	BIT
yes/no	boolean or java.lang.Boolean	CHAR(1) ('Y' or 'N')
true/false	boolean or java.lang.Boolean	CHAR(1) ('T' or 'F')

Date and Time Types

Mapping type	Java type	ANSI SQL Type
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

Binary and Large Object Types

Mapping type	Java type	ANSI SQL Type
binary	byte[]	VARBINARY (or BLOB)
text	java.lang.String	CLOB
serializable	any Java class that implements java.io.Serializable	VARBINARY (or BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

JDK-related Types

Mapping type	Java type	ANSI SQL Type
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

GTU-PAPER->What is HQL? How does it differ from SQL? Give its advantages.

- Hibernate Query Language HQL is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties.
- The Hibernate Query Language (HQL) is an easy-to-learn and powerful query language designed as an object-oriented extension to SQL.
- HQL queries are case insensitive; however, the names of java classes and properties are case sensitive.If HQL is used in an application define a query for a database, the Hibernate framework automatically generates the SQL query and execute it.
- HQL can also be used to retrieve objects from database through O/R mapping by performing the following tasks:
  - Apply restrictions to properties of objects
  - Arrange the results returned by a query by using the order by clause
  - Paginate the results
  - Aggregate the records by using group by and having clauses
  - Use Joins
  - Create user-defined functions
  - Execute subqueries
- Following are some of the reasons why HQL is preferred over SQL:
  - Provides full support for relation operations
  - Returns results as objects
  - Support polymorphic queries
  - Easy to learn and use
  - Supports for advanced features
  - Provides database independency

- HQL consists of the following elements:
  - Clauses
    - From : from object [as object_alias]
    - Select : select [object.property]
    - Where : where condition
    - Order by : order by obj1.property1 [asc|desc] [obj2.property2]
    - Group by : Group by obj1.property1,obj1.property2
  - Association and Joins
    - Inner join
    - Left outer join
    - Right outer join
    - Full join
  - Aggregate functions
    - avg(…)
    - sum(…)
    - min(…)
    - max(…)
    - count(…)
  - Expressions
    - Mathematical operators : +,-,*,/
    - Binary comparison operators : =,>=,<=,<>,!=
    - Logical operators : and,or,not
    - String concatenation : ||
  - Subqueries
  - A query within query is known as subquery and is surrounded by parentheses.
- **FROM Clause :**
  You will use FROM clause if you want to load a complete persistent objects into memory. Following is the simple syntax of using FROM clause:

  ```
  String hql = "FROM Employee";
  Query query = session.createQuery(hql);
  List results = query.list()
  ```
- ***SELECT Clause :***
  The SELECT clause provides more control over the result set than the from clause. If you want to obtain few properties of objects instead of

the complete object, use the SELECT clause.

```
String hql = "SELECT E.firstName FROM Employee E"; Query query = session.createQuery(hql);
List results = query.list();
```

- ***ORDER BY Clause :***
  To sort your HQL query's results, you will need to use the ORDER BY clause.

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC"; Query query = session.createQuery(hql);
List results = query.list();
```

- ***USING NAMED Parameters :***

  Hibernate supports named parameters in its HQL queries.This makes writing HQL queries that accept input from the user.

  ```
  String hql = "FROM Employee E WHERE E.id = :employee_id";
  Query query = session.createQuery(hql);
  query.setParameter("employee_id",10); List results = query.list();
  ```

- ***UPDATE Clause :***

  The UPDATE clause can be used to update one or more properties of an one or more objects.

  ```
  String hql = "UPDATE Employee set salary = :salary WHERE id = :employee_id"; Query query = session.createQuery(hql);
  query.setParameter("salary", 1000);
  query.setParameter("e
  mployee_id", 10); int
  result =
  query.executeUpdate(
  );
  System.out.println("Rows affected: " + result);
  ```

- ***Pagination Using Query :***

  Query setFirstResult - This method takes an integer that represents the first row in your result set, starting with row 0.

  Query setMaxResult - This method tells Hibernate to retrieve a fixed number maxResults of objects.

```
String hql = "FROM Employee";
Query query =
session.createQuery(h
ql);
query.setFirstResult(1
);
query.setMaxResults(
10);
List results = query.list();
```

Hibernate Configuration:

- Hibernate requires to know in advance where to find the mapping information that defines how your Java classes relate to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters.
- All such information is usually supplied as a standard Java properties file called hibernate.properties, or as an XML file namedhibernate.cfg.xml.

Following are Hibernate properties :

Sr.No.	Properties & Description
1	**hibernate.dialect** This property makes Hibernate generate the appropriate SQL for the chosen database.
2	**hibernate.connection.driver_class** The JDBC driver class.
3	**hibernate.connection.url** The JDBC URL to the database instance.
4	**hibernate.connection.username** The database username.

5	**hibernate.connection.password** The database password.
6	**hibernate.connection.pool_size** Limits the number of connections waiting in the Hibernate database connection pool.
7	**hibernate.hbm2ddl.auto** Allow automatic table creation

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">sun.jdbc.odbc.JdbcOdbcDriver</property>
    <property name="hibernate.connection.url">jdbc:odbc:test</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.hbm2ddl.auto">create</property>
    <mapping resource="hibernate.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Hibernate O/R Mapping:

- ORM is technique to map object-oriented data with relational data.
- In other words, it is used to convert the datatype supported in object oriented programming language to a datatype supported by a database.
- ORM is also known as O/RM & O/R mapping.
- Mapping should be in the format that can define the mapping of the
  - classes with tables
  - properties with columns
  - foreign key with associations
  - and SQL types with Java types.
- Mapping can be done with Oracle, DB2, MySql, Sybase and any other relational databases.
- Mapping will be done in the form of XML Document.
- In complex applications, a class can be inherited by other classes, which may lead to the problem of class mismatching. To overcome such problem, the following approach can be used.
  - **Table-per-class-hierarchy:** Removes polymorphism and inheritance relationship completely from the relational model
  - **Table-per-subclass(normalized mapping):** De-Normalizes the relational model and enables polymorphism.
  - **Table-per-concrete-class:** Represents inheritance relationship as foreign key relationships.
- Multiple objects can be mapped to a single row. Polymorphic associations for the 3 strategies are as follows,
  - **Many-to-one:** Serves as an association in which an object reference is mapped to a foreign key association.
  - **One-to-many:** Serves as an association in which a collection of objects is mapped to a foreign key association.
  - **Many-to-many:** Serves as an association in which a collection of objects is transparently mapped to a match table.
- The important elements of the Hibernate mapping file are as follows.

- **DOCTYPE:** Refers to the Hibernate mapping Document Type Declaration (DTD) that should be declared in every mapping file for syntactic validation of the XML.
- **<hibernate-mapping> element:** Refers as the first or root element of Hibernate, inside <hibernate-mapping> tag any number of class may be present.
- **<class> element:** Maps a class object with its corresponding entity in the database.
- **<id> element :** Serves as a unique identifier used to map primary key column of the database table.
- **<generator> element:** Helps in generation of the primary key for the new record, following are some of the commonly used generators.
    - Increment , Sequence, Assigned, Identity, Hilo, Native

- **<property> element:** Defines standard Java attributes and database schema.

Example:

This XML file tells what instance variable of **studentdetail** class is to be mapped to what column of table **school**.

The file should have an extension **.hbm.xml**. The name is given as Student just to remember **hibernate.hbm.xml** file belongs to class Student, but for it can be any name.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
   <class name="com.Studetail" table="stud1">
      <id name="sid" column="STUD_ID" type="integer">
         <generator class="assigned"/>
      </id>
      <property name="name" column="STUD_Name" type="string"/>


   </class>
```

</hibernate-mapping></hibernate-mapping> This is the heart of the entire Hibernate where programmer writes instructions to Hibernate how it should behave with **Java variables** and **database columns**. There must be a primary key in the table and the primary key here is **STUD_ID** corresponding to **sid** in the Student bean class. Let us go in detail.

ELEMENT NAME	FUNCTIONALITY
<HIBERNATE-MAPPING>	It is the root element of the above XML mapping file. <class> element is the child element of it.
<CLASS>	This element contains all the mapping information of which Java bean variable (or field or property) is to be mapped to which database column. The "name" attribute should be given the name of the Java bean class (here, it is **studentdetail**) and "table" attribute should be given the name of the database table (here, it is **school**).
<ID>	This element gives Hibernate the primary key information of the table. "name" attribute gives the name of the Java bean variable. The "column" attribute gives the table column name. Here the value of **sid** is to be inserted into **stud_id** column of table **school**. **stud_id** works as primary key. "type" attribute is given value as **integer**. The **integer** value is the data type of Hibernate and not of Java. The **integer** value converts **sid int** value to database specific SQL column type (in Oracle, it is number).
<GENERATOR>	This describes the primary key assignment. The primary key value can be assigned by the programmer himself or can be asked the Hibernate to assign a value by itself as per its algorithm. Here, the value assigned tells Hibernate that primary key value is assigned by the programmer himself. How to tell the Hibernate to generate itself (where programmer will not give), we will see later.

<PROPERTY>	This element tells Hibernate which bean field to be mapped to which table column. The "name" attribute should be given the name of the bean field (or instance variable) name and "column" attribute should be given the name of the column in datable table. In the first property, the value of **sname** of **studentdetail** class should be inserted into the**stud_name** column of **school** table. The "type" attribute value refers Hibernate string (not Java String) which at runtime converts **sname**String value to database specific column type (in Oracle, it is varchar).	

The type **integer**, **string**, **double** and **date** are the data types of Hibernate not of Java which at runtime converted to database specific (here, it is Oracle) column types.

Hibernate Simple Example

Note: This notes on Hibernate Example gives the basic steps of writing a Hibernate program. Explained in simple steps.

Here, a **Student** record is inserted into database table using Hibernate. All the code is explained very clearly with the relevant **XML** files. The execution part of this program is given separately in Simple Hibernate Program – Step-by-step Execution with screenshots. It advised to read the Hibernate Tutorial before going into this program.
First let us go for a simple program where a **Student** object is inserted (persisted) into MS ACCESS database table **school**.

STUDETAIL CLASS		STUDENT TABLE	
JAVA INSTANCE VARIABLE	TYPE	TABLE COLUMN NAME	TYPE
Sid	int	STUD_ID (primary key)	number(4)
Sname	String	STUD_NAME	VARCHAR2(15)

Now you are ready with school table and further require 4 programs.

1. **studentdetail.java – Java class written with JavaBean syntax.**
2. **hibernate.cfg.xml – A Configuration file**
3. **hibernate.hbm.xml – A Mapping file.**
4. **StudentClient.java – Java Client program that inserts (or stores or persists) Student objects in database.**

Now let us see each of the above 4 files and discuss line-wise.

1. Java program Student.java with JavaBean syntax with setter and getter methods (Persistent class)

The variables of this Java program correspond to columns in a database table. We create an object **Student** class, set the properties and persist (storing) in database table **school**.

```
package com;


public class Studetail {
  public  int sid;
    public String name;
    public Studetail()
    {

    }
    public void setsid(int sid)
    {
       this.sid=sid;
    }
    public int getsid()
    {
       return (sid);
    }
    public void setname(String name)
    {
```

```
        this.name=name;
    }
    public String getname()
    {
        return (name);
    }

}
```

Now you are ready with the class **studentdetail** and table **school**.
It requires two XML files – **hibernate.cfg.xml** and **hibernate.hbm.xml**.

2. Configuration File – hibernate.cfg.xml

This file comes with Hibernate software distribution. Just override as per your convenience. This XML file mainly tells Hibernate what database you are using to insert the data along with user name and password etc. The file should have an extension **.cfg.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">sun.jdbc.odbc.JdbcOdbcDriver</property>
    <property name="hibernate.connection.url">jdbc:odbc:test</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.hbm2ddl.auto">create</property>
    <mapping resource="hibernate.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

The syntax is almost self-explanatory but for a few. **Dialect** tells the name of the database we are using. Hibernate supports Oracle. The other databases Hibernate supports are given at the end of this notes.

<session-factory> is responsible to connect to database. Programmer should provide all the required information in the form of XML tags. The first five **<property>** elements give the database specific information including the driver being used. The information of this XML file is used by **SessionFactory** object in the client program to connect to database. If the same Java bean values (called as properties) are to be inserted in multiple databases (like one table in Oracle and one table in MS-Access etc.), multiple **SessionFactory** objects are required.

show_sql shows on the Console (of MyEclipse), the SQL statements created by **hibernate.hbm.xml** is the XML file where mappings between Java variables and database columns are given.

3. Mapping File – hibernate.hbm.xml

This XML file tells what instance variable of **studentdetail** class is to be mapped to what column of table **school**.

The file should have an extension **.hbm.xml**. The name is given as Student just to remember **hibernate.hbm.xml** file belongs to class Student, but for it can be any name.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
   <class name="com.Studetail" table="stud1">
      <id name="sid" column="STUD_ID" type="integer">
         <generator class="assigned"/>
      </id>
      <property name="name" column="STUD_Name" type="string"/>


   </class>
</hibernate-mapping></hibernate-mapping>
```

This is the heart of the entire Hibernate where programmer writes instructions to Hibernate how it should behave with **Java variables** and **database columns**. There must be a primary

key in the table and the primary key here is **STUD_ID** corresponding to **sid** in the Student bean class. Let us go in detail.

ELEMENT NAME	FUNCTIONALITY
<HIBERNATE-MAPPING>	It is the root element of the above XML mapping file. <class> element is the child element of it.
<CLASS>	This element contains all the mapping information of which Java bean variable (or field or property) is to be mapped to which database column. The "name" attribute should be given the name of the Java bean class (here, it is **studentdetail**) and "table" attribute should be given the name of the database table (here, it is **school**).
<ID>	This element gives Hibernate the primary key information of the table. "name" attribute gives the name of the Java bean variable. The "column" attribute gives the table column name. Here the value of **sid** is to be inserted into **stud_id** column of table **school**. **stud_id** works as primary key. "type" attribute is given value as **integer**. The **integer** value is the data type of Hibernate and not of Java. The **integer** value converts **sid int** value to database specific SQL column type (in Oracle, it is number).
<GENERATOR>	This describes the primary key assignment. The primary key value can be assigned by the programmer himself or can be asked the Hibernate to assign a value by itself as per its algorithm. Here, the value assigned tells Hibernate that primary key value is assigned by the programmer himself. How to tell the Hibernate to generate itself (where programmer will not give), we will see later.

<PROPERTY>	This element tells Hibernate which bean field to be mapped to which table column. The "name" attribute should be given the name of the bean field (or instance variable) name and "column" attribute should be given the name of the column in datable table. In the first property, the value of **sname** of **studentdetail** class should be inserted into the**stud_name** column of **school** table. The "type" attribute value refers Hibernate string (not Java String) which at runtime converts **sname**String value to database specific column type (in Oracle, it is varchar).	

The type **integer**, **string**, **double** and **date** are the data types of Hibernate not of Java which at runtime converted to database specific (here, it is Oracle) column types.

4. Client program – StudentClient.java

Finally we write a client program which when executed inserts records in the database table.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package com;


import org.hibernate.*;
import org.hibernate.cfg.*;
public class demo {
   public static void main(String []args) throws Exception
   {
```

```
Configuration c=new Configuration();
c.configure("/hibernate.cfg.xml");
SessionFactory sf=c.buildSessionFactory();
Session s=sf.openSession();
Transaction tx=s.beginTransaction();
try{
   Studetail s1=new Studetail();
   s1.setsid(1);
   s1.setname("nirali");
   Studetail s2=new Studetail();
   s2.setsid(2);
   s2.setname("diya");
   s.save(s1);
   s.save(s2);
   s.flush();
   tx.commit();
   System.out.println("record inserted");
}
catch(Exception e)
{
   tx.rollback();
}
```

```
    }
}
```

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

The starting point of the client program is creating an object of **Configuration** class.**Configuration** object, here it is **c**, job is to load the **hibernate.cfg.xml** file, read the database particulars and return these particulars to **SessionFactory** object **sf**.

```
Session s = sf.openSession();
```

The **Session** object is equivalent to **Connection** object of JDBC. Any data to be persisted in the database is passed to **Session** object (say, save()), here it is **s**.
Creation of **Transaction** object **tx** is required for all save, delete and update operations. Select statements (reading records) do not require. **Transaction** object dictates the boundaries of transaction with **beginTransaction()** and **commit()** (or **rollback()**) methods.

```
Student std1=new Student();
std1.setSid(100);
std1.setSname("S N Rao");
std1.setSmarks(78);
std1.setSjoindate(new Date());
```

Two **Student** objects **std1** and **std2** are created and properties are set with **setXXX()** methods as declared in Student bean program.

```
s.save(std1);
s.save(std2);
```

The **save()**, **update()** and **delete()** methods of **Session** class do not insert data immediately. Instead, these methods write **addBatch()** statements and keep ready for execution. The **flush()**method internally writes one **executeBatch()** statement and executes all the **addBatch()**statements. This style increases considerable performance.

```
tx.commit();
```

The **commit()** method of **Transaction** does commit operation on the statement executed earlier with **flush()** method.

class attributes (in hibernate.cfg.xml file)

Following are the most frequently used values.

1. **increment:** It increments itself and generates ID of type short, int or long.
2. **identity:** It is specific for DB2, MySQL server, Sybase etc. The generated type may be short, int or long.
3. **sequence:** To generate an ID, the database uses a sequence execution. The generated type may be short, int or long.
4. **hilo:** High-low generator internally uses a hi/lo algorithm to generate identifiers of data type short, int and long.
5. **native:** It takes some algorithm like sequence, identity or hilo specific to the database.
6. **assigned:** Here, programmer should assign the ID for himself.

7. foreign: It uses an ID of another associated object; generally used in one-to-one association.

Dialects (databases) supported by Hibernate

Hibernate supports following databases.

SQL Dialects in Hibernate

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

RDBMS	Dialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle9i	org.hibernate.dialect.Oracle9iDialect
Oracle10g	org.hibernate.dialect.Oracle10gDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
Firebird	org.hibernate.dialect.FirebirdDialect

Annotated Class Example:

As I mentioned above while working with Hibernate Annotation all the metadata is clubbed into the POJO java file along with the code this helps the user to understand the table structure and POJO simultaneously during the development.

Consider we are going to use following EMPLOYEE table to store our objects:

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Following is the mapping of Employee class with annotations to map objects with the defined EMPLOYEE table:

```
import javax.persistence.*;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {
   @Id @GeneratedValue
   @Column(name = "id")
   private int id;

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

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

   @Column(name = "salary")
   private int salary;
```

```
  public Employee() {}
  public int getId() {
    return id;
  }
  public void setId( int id ) {
    this.id = id;
  }
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName( String first_name ) {
    this.firstName = first_name;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName( String last_name ) {
    this.lastName = last_name;
  }
  public int getSalary() {
    return salary;
  }
  public void setSalary( int salary ) {
    this.salary = salary;
  }
}
```

Hibernate detects that the @Id annotation is on a field and assumes that it should access properties on an object directly through fields at runtime. If you

placed the @Id annotation on the getId() method, you would enable access to properties through getter and setter methods by default. Hence, all other annotations are also placed on either fields or getter methods, following the selected strategy. Following section will explain the annotations used in the above class.

@Entity Annotation:

The EJB 3 standard annotations are contained in the **javax.persistence**package, so we import this package as the first step. Second we used the**@Entity** annotation to the Employee class which marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

@Table Annotation:

The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database.

The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table. For now we are using just table name which is EMPLOYEE.

@Id and @GeneratedValue Annotations:

Each entity bean will have a primary key, which you annotate on the class with the **@Id** annotation. The primary key can be a single field or a combination of multiple fields depending on your table structure.

By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used but you can override this by applying the **@GeneratedValue** annotation which takes two parameters **strategy** and **generator** which I'm not going to discuss here, so let us use only default the default key generation strategy. Letting Hibernate determine which generator type to use makes your code portable between different databases.

@Column Annotation:

The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:

- **name** attribute permits the name of the column to be explicitly specified.
- **length** attribute permits the size of the column used to map a value particularly for a String value.
- **nullable** attribute permits the column to be marked NOT NULL when the schema is generated.

Note: If you want to write a HQL query in program, you can write as follows:

```
Query qry = session.createQuery("delete from Product p
where p.productId=:var");
        qry.setParameter("var",110);
        int res = qry.executeUpdate();
```

```
Query qry = session.createQuery("delete from Product p
where p.productId=?");
        qry.setParameter(0,110);
        int res = qry.executeUpdate();
```

```
Query query = session.createQuery("from EMPLOYEE");
    java.util.List list = query.list();
    System.out.println(list);
```