Hibernate Interview Questions

Version 1.0

Sanjeev Kumar Singh
6/1/2011

1. **What is ORM?**

Ans. ORM is Object Relational Mapping. It provides a way to map a Java data model to a corresponding database model in both directions. The basic purpose of ORM is to allow an application written in an object oriented language to deal with the information it manipulates in terms of objects, rather than in terms of database-specific concepts such as rows, columns and tables. If we setup this mapping with Hibernate, Java Persistence API or another ORM framework, we don't have to worry any longer about the low level details of database interactions. Basically we define a mapping between Java classes and database tables or class members and table columns and the ORM mapper takes care of the communication between your Java code and an underlying database.

2. **What does an ORM solution comprises of?**

1. It should have an API for performing basic CRUD (Create, Read, Update, Delete) operations on objects of persistent classes
2. Should have a language or an API for specifying queries
3. An ability for specifying mapping metadata
4. Optimization facilities (It should have a technique for ORM implementation to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions)

3. **What are the different levels of ORM quality?**

There are four levels defined for ORM quality.
1. Pure relational
2. Light object mapping
3. Medium object mapping
4. Full object mapping

4. **What is a pure relational ORM?**

The entire application, including the user interface, is designed around the relational Model and SQL-based relational operations.

5. **What is a meant by light object mapping?**

The entities are represented as classes that are mapped manually to the relational tables. The code is hidden from the business logic using specific design patterns. This approach is successful for applications with a less number of entities, or applications with common, metadata-driven data models. This approach is most known to all.

6. **What is a meant by medium object mapping?**

The application is designed around an object model. The SQL code is generated at build time. And the associations between objects are supported by the persistence mechanism, and queries are specified using an object-oriented expression language. This is best suited for medium-sized applications with some complex transactions. Used when the mapping exceeds 25 different database products at a time.

7. **What is meant by full object mapping?**

Full object mapping supports sophisticated object modeling: composition, inheritance, polymorphism and persistence. The persistence layer implements transparent persistence; persistent classes do not inherit any special base class or have to implement a special interface. Efficient fetching strategies and caching strategies are implemented transparently to the application.

8. **What is Hibernate?**

Hibernate is a pure Java object-relational mapping (ORM) and persistence framework that allows us to map plain old Java objects to relational database tables using (XML) configuration files. Its main purpose is to relieve the developer from a significant amount of relational data persistence-related programming tasks.

9. **What are the benefits of ORM and Hibernate?**

- **Improved productivity**
  - High-level object-oriented API
  - Less Java code to write
  - No SQL to write
- **Improved performance**
  - Sophisticated caching
  - Lazy loading
  - Eager loading
- **Improved maintainability**
  - A lot less code to write
- **Improved portability**
  - ORM framework generates database-specific SQL for you

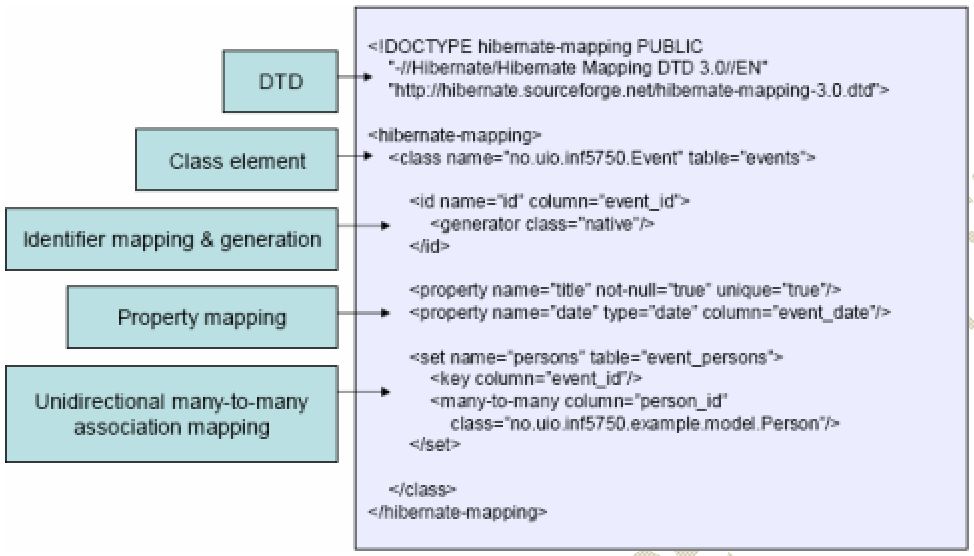
How does hibernate code looks like?

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
MyPersistanceClass mpc = new MyPersistanceClass ("Sample App");
session.save(mpc);
tx.commit();
session.close();
```

The Session and Transaction are the interfaces provided by hibernate. There are many other interfaces besides this.

10. **What is a hibernate xml mapping document and how does it look like?**

This xml represent the object/relation mapping. Hibernate mapping file tells Hibernate which tables and columns to use to load and store objects. Typical mapping file look as follows:


DTD
Class element
Identifier mapping & generation
Property mapping
Unidirectional many-to-many association mapping

```
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="no.uio.inf5750.Event" table="events">

    <id name="id" column="event_id">
      <generator class="native"/>
    </id>

    <property name="title" not-null="true" unique="true"/>
    <property name="date" type="date" column="event_date"/>

    <set name="persons" table="event_persons">
      <key column="event_id"/>
      <many-to-many column="person_id"
        class="no.uio.inf5750.example.model.Person"/>
    </set>

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

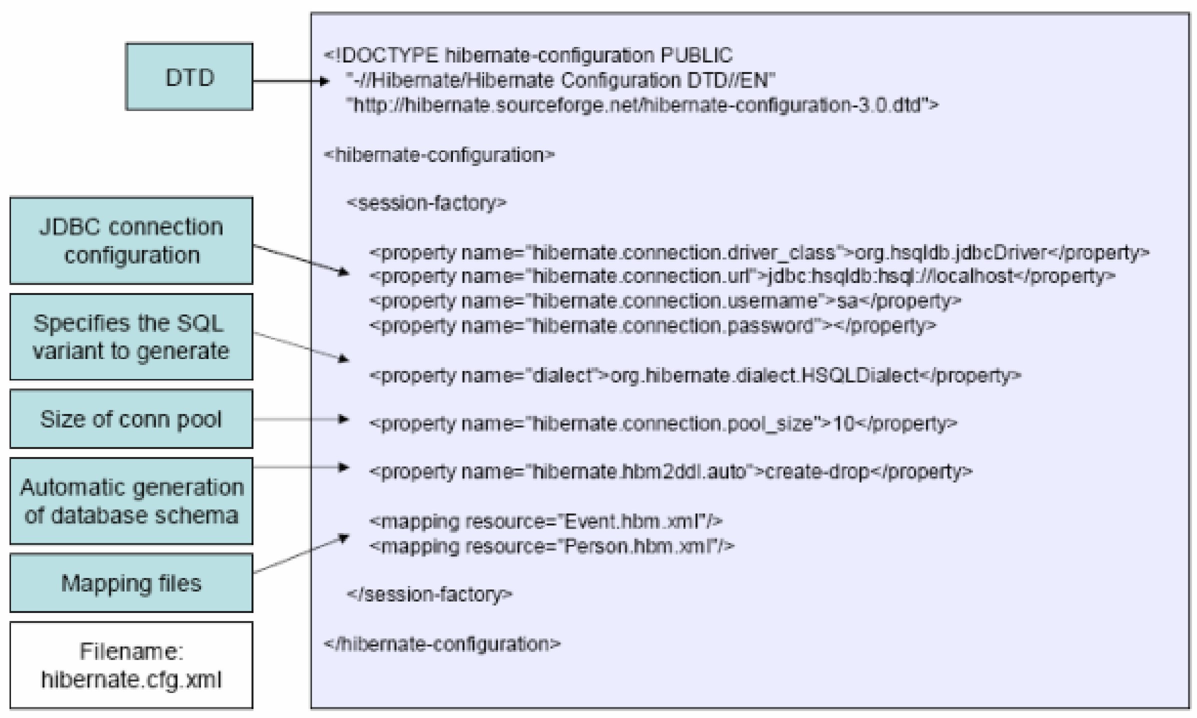
11. **What are the most common methods of Hibernate configuration?**

The most common methods of Hibernate configuration are:

- Programmatic configuration
- XML configuration (hibernate.cfg.xml)

12. **What are the important tags of hibernate.cfg.xml?**

- Following are the important tags of hibernate.cfg.xml


DTD
JDBC connection configuration
Specifies the SQL variant to generate
Size of conn pool
Automatic generation of database schema
Mapping files
Filename: hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

  <session-factory>

    <property name="hibernate.connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="hibernate.connection.url">jdbc:hsqldb:hsql://localhost</property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password"></property>

    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

    <property name="hibernate.connection.pool_size">10</property>

    <property name="hibernate.hbm2ddl.auto">create-drop</property>

    <mapping resource="Event.hbm.xml"/>
    <mapping resource="Person.hbm.xml"/>

  </session-factory>

</hibernate-configuration>
```

13. **What are the Core interfaces are of Hibernate framework?**

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

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. 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. The Criteria provides a provision for conditional search over the results set. One can retrieve entities by composing criterion objects. The Session is a factory for Criteria. Criterion instances are usually obtained via the factory methods on Restrictions. Query represents object oriented representation of a Hibernate query.

Query instance is obtained by calling

Session.createQuery().

14. **What role does the Session interface play in Hibernate?**

It is single threaded short lived object representing the conversation between your application and database.

Session session = sessionFactory.openSession()

Role of Session interface are as follows.

- Wraps a JDBC connection
- Factory for Transaction
- Holds a mandatory (first-level) cache of persistent objects

15. **What role does the SessionFactory interface play in Hibernate?**

The application obtains Session instances from a SessionFactory. There is typically a single SessionFactory for the whole application application

Role of SessinFactory are as follows

- Caches generate SQL statements and other mapping metadata that Hibernate uses at runtime.
- It also holds cached data that has been read in one unit of work and may be reused in a future unit of work

  SessionFactory sessionFactory = configuration.buildSessionFactory();

16. **What is the general flow of Hibernate communication with RDBMS?**

The general flow of Hibernate communication with RDBMS is :

- Load the Hibernate configuration file and create configuration object. It will automatically load all hbm mapping files
- Create session factory from configuration object
- Get one session from this session factory
- Create HQL Query
- Execute query to get list containing Java objects

17. **What is Hibernate Query Language (HQL)?**

Hibernate offers a query language that embodies a very powerful and flexible mechanism to query, store, update, and retrieve objects from a database. This language, the Hibernate query Language (HQL), is an object-oriented extension to SQL.

How do you map Java Objects with Database tables?

- First we need to write Java domain objects (beans with setter and getter).
- Write hbm.xml, where we map java class to table and database columns to Java class variables.

Example :

```
<hibernate-mapping>
  <class name="com.test.User"  table="user">
   <property  column="USER_NAME" length="255"
   name="userName" not-null="true"  type="java.lang.String"/>
  <property  column="USER_PASSWORD" length="255"
   name="userPassword" not-null="true"  type="java.lang.String"/>
 </class>
</hibernate-mapping>
```

18. **What is lazy fetching in Hibernate? With Example.**

Lazy fetching decides whether to load child objects while loading the Parent Object. You need to do this setting respective hibernate mapping file of the parent class. Lazy = true (means not to load child) - By default the lazy loading of the child objects is true. This make sure that the child objects are not loaded unless they are explicitly invoked in the application by calling getChild() method on parent. In this case hibernate issues a fresh database call to load the child when getChild () is actually called on the Parent object. But in some cases you do need to load the child objects when parent is loaded. Just make the lazy=false and hibernate will load the child when parent is loaded from the database.

Example:
If you have a TABLE? EMPLOYEE mapped to Employee object and contains set of Address objects.

```
Parent Class : Employee class
Child class : Address Class
public class Employee {
private Set address = new HashSet(); // contains set of child Address objects
public Set getAddress () {
return address;
}
public void setAddresss(Set address) {
this. address = address;
}
}
In the Employee.hbm.xml file
<set name="address" inverse="true" cascade="delete" lazy="false">
      <key column="a_id" />
<one-to-many class="beans Address"/>
</set>
```

In the above configuration - If lazy="false" : - when you load the Employee object that time child object Adress is also loaded and set to setAddresss() method. If you call employee.getAdress() then loaded data returns.No fresh database call. If lazy="true":- This the default configuration. If you don't mention then hibernate consider lazy=true. When you load the Employee object that time child object Address is not loaded. You need extra call to data base to get address objects. If you call employee.getAdress() then that time database query fires and return results. Fresh database call.

19. **What is c3p0? How To configure the C3P0 connection pool?**

It is a Connection pooling API. We need to add the library C3P0 jar to our application lib. Then we have to configure it in our hibernate configuration file.

```
<!-- configuration pool via c3p0-->
<property name="c3p0.acquire_increment">1</property>
<property name="c3p0.idle_test_period">100</property> <!-- seconds -->
<property name="c3p0.max_size">100</property>
<property name="c3p0.max_statements">0</property>
<property name="c3p0.min_size">10</property>
<property name="c3p0.timeout">100</property> <!-- seconds -->
<!-- DEPRECATED very expensive property name="c3p0.validate>-->
```

20. **What's the difference between load() and get()?**

load() vs. get() :-

load()	**get()**
Only use the load() method if you are sure that the object exists.	If you are not sure that the object exists, then use one of the get() methods.
load() method will throw an exception if the unique id is not found in the database.	get() method will return null if the unique id is not found in the database.
load() just returns a proxy by default and database won't be hit until the proxy is first invoked.	get() will hit the database immediately.

21. **Difference between session.save () , session.saveOrUpdate() and session.persist()?**

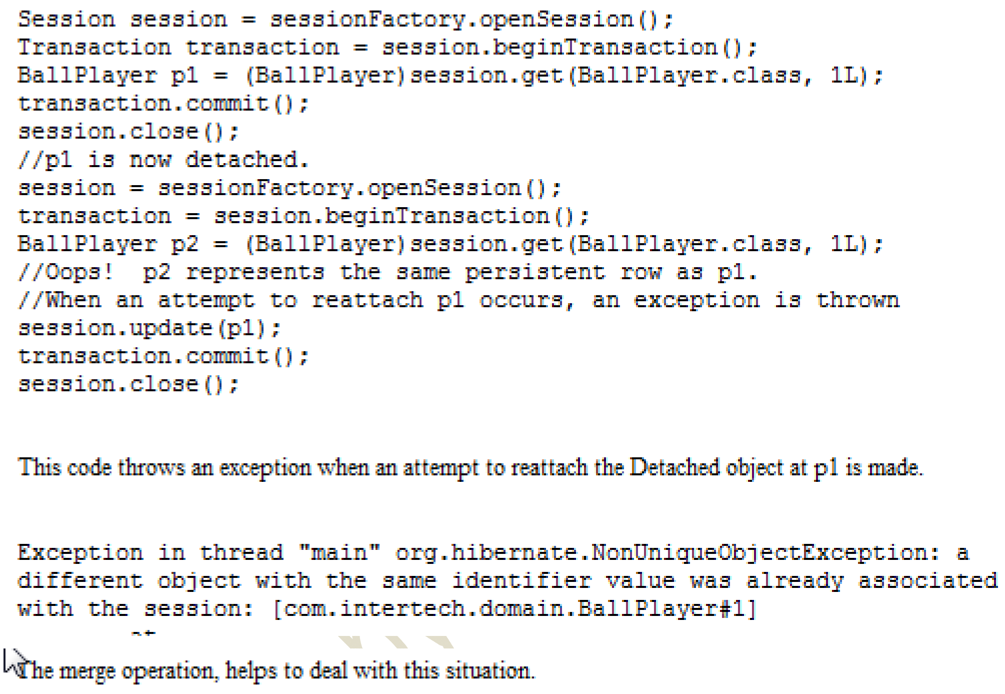
session.save() : Save does an insert and will fail if the primary key is already persistent.
session.saveOrUpdate() : It does a select first to determine if it needs to do an insert or an update. Insert data if primary key not exist otherwise update data. It returns void.
session.persist() : Does the same like session.save(). But session.save () return Serializable object but session.persist() return void. session.save() returns the generated identifier (Serializable object) and session.persist() doesn't.

For Example :

If we do :-
System.out.println(session.save(question));
This will print the generated primary key.
if you do :-
System.out.println(session.persist(question));
Compile time error because session.persist() return void.

22. **What is the difference between and merge and update?**

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

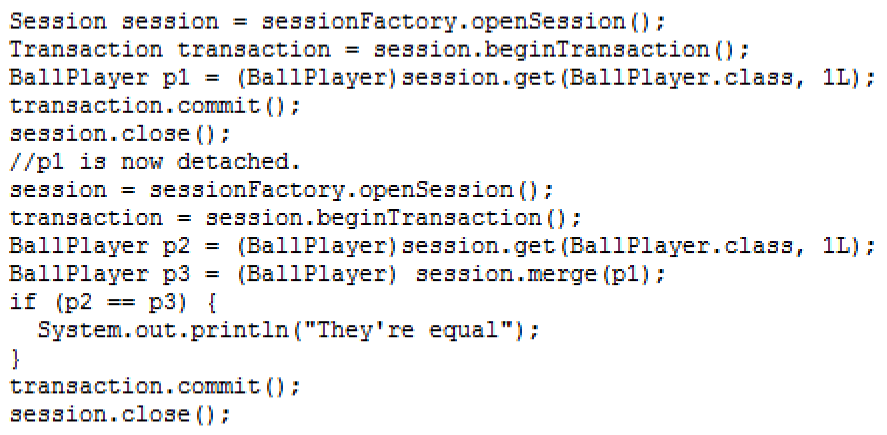


```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
BallPlayer p1 = (BallPlayer)session.get(BallPlayer.class, 1L);
transaction.commit();
session.close();
//p1 is now detached.
session = sessionFactory.openSession();
transaction = session.beginTransaction();
BallPlayer p2 = (BallPlayer)session.get(BallPlayer.class, 1L);
//Oops!  p2 represents the same persistent row as p1.
//When an attempt to reattach p1 occurs, an exception is thrown
session.update(p1);
transaction.commit();
session.close();
```

This code throws an exception when an attempt to reattach the Detached object at p1 is made.

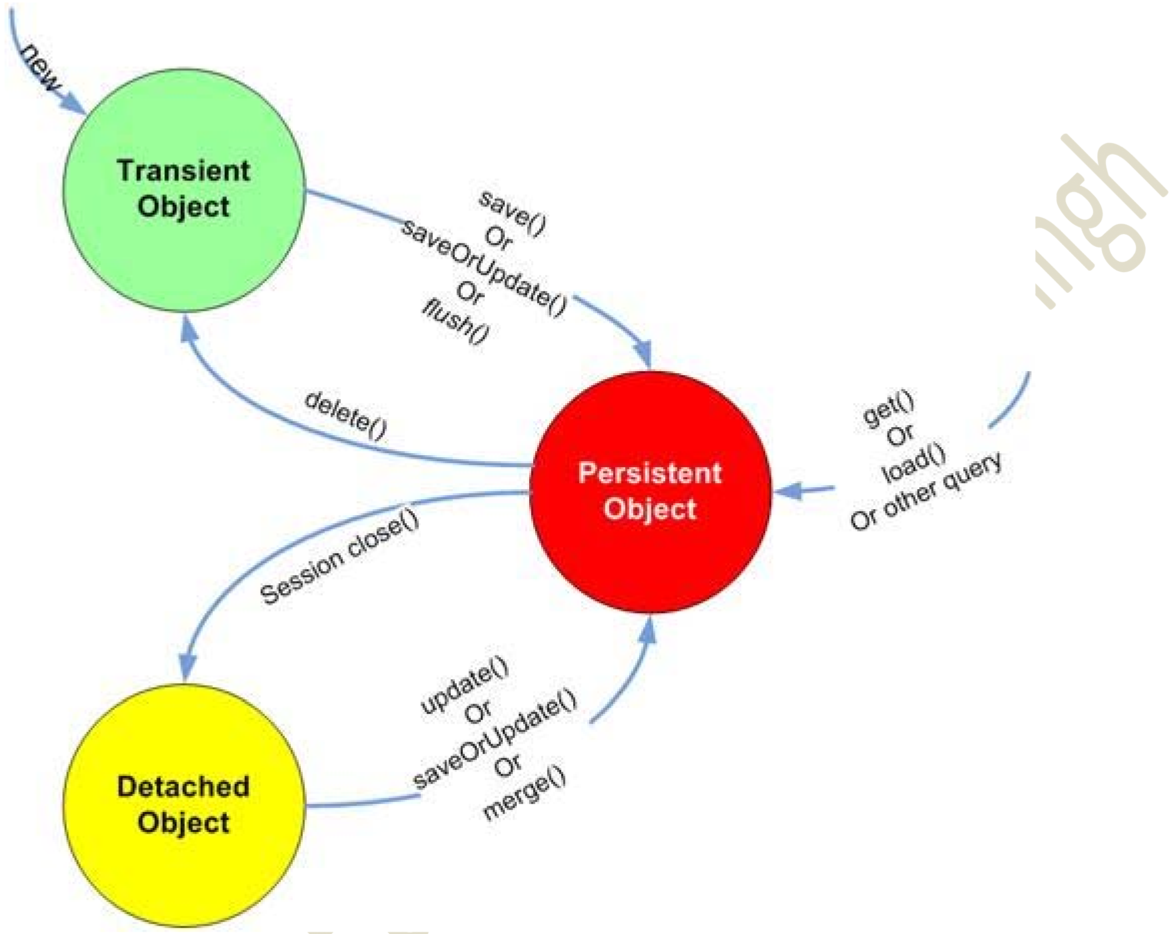
```
Exception in thread "main" org.hibernate.NonUniqueObjectException: a
different object with the same identifier value was already associated
with the session: [com.intertech.domain.BallPlayer#1]
```

The merge operation, helps to deal with this situation.



```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
BallPlayer p1 = (BallPlayer)session.get(BallPlayer.class, 1L);
transaction.commit();
session.close();
//p1 is now detached.
session = sessionFactory.openSession();
transaction = session.beginTransaction();
BallPlayer p2 = (BallPlayer)session.get(BallPlayer.class, 1L);
BallPlayer p3 = (BallPlayer) session.merge(p1);
if (p2 == p3) {
  System.out.println("They're equal");
}
transaction.commit();
session.close();
```

The merge () method is a little complex and works differently depending on what is in/out of the persistence context. Hibernate will first check whether a Persistent instance of that type already exists in the persistent context. It uses the object identifiers to check on this existence. If another instance exists, it copies the state of the Detached object (p1 above) into the existing Persistence object (p2 above). If no other instance exists, Hibernate just reattaches the Detached object.


new
Transient Object
save() Or saveOrUpdate() Or flush()
delete()
Persistent Object
get() Or load() Or other query
Session close()
Detached Object
update() Or saveOrUpdate() Or merge()

18.How do you define sequence generated primary key in hibernate?

Using <generator> tag.
Example:-

```
<id column="USER_ID" name="id" type="java.lang.Long">
  <generator class="sequence">
    <param name="table">SEQUENCE_NAME</param>
  <generator>
</id>
```

23. **Define cascade and inverse option in one-many mapping?**

Cascade is a convenient feature to save the lines of code needed to manage the state of the other side manually. It is used whenever we have a parent child relationship if parent record is changed then child record should also be changed.

1) cascade="none", the default, tells Hibernate to ignore the association.
2) cascade="save-update" tells Hibernate to navigate the association when the transaction is committed and when an object is passed to save() or update() and save newly instantiated transient instances and persist changes to detached instances.
3) cascade="delete" tells Hibernate to navigate the association and delete persistent instances when an object is passed to delete().
4) cascade="all" means to cascade both save-update and delete, as well as calls to evict and lock.
5) cascade="all-delete-orphan" means the same as cascade="all" but, in addition, Hibernate deletes any persistent entity instance that has been removed (dereferenced) from the association (for example, from a collection).
6) cascade="delete-orphan" Hibernate will delete any persistent entity instance that has been removed (dereferenced) from the association (for example, from a collection).

With save-update cascade

The **cascade="save-update"** is declared in 'stockDailyRecords' to enable the save-update cascade effect.

```
<!-- Stock.hbm.xml -->
<set name="stockDailyRecords" cascade="save-update" table="stock_daily_record"...>
      <key>
            <column name="STOCK_ID" not-null="true" />
      </key>
      <one-to-many class="com.mkyong.common.StockDailyRecord" />
</set>
```

```
Stock stock = new Stock();
StockDailyRecord stockDailyRecords = new StockDailyRecord();
//set the stock and stockDailyRecords  data

stockDailyRecords.setStock(stock);
stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);
```

The code **session.save(stockDailyRecords);** is no longer required, when you save the 'Stock', it will "cascade" the save operation to it's referenced 'stockDailyRecords' and save both into database automatically.

No delete cascade

You need to loop all the 'stockDailyRecords' and delete it one by one.

```
Query q = session.createQuery("from Stock where stockCode = :stockCode ");
q.setParameter("stockCode", "4715");
Stock stock = (Stock)q.list().get(0);

for (StockDailyRecord sdr : stock.getStockDailyRecords()){
         session.delete(sdr);
}
 session.delete(stock);
```

Output

```
Hibernate:
    delete from mkyong.stock_daily_record
    where DAILY_RECORD_ID=?

Hibernate:
    delete from mkyong.stock
    where STOCK_ID=?
```

With delete cascade

The **cascade="delete"** is declared in 'stockDailyRecords' to enable the delete cascade effect. When you delete the 'Stock', all its reference 'stockDailyRecords' will be deleted automatically.

```
<!-- Stock.hbm.xml -->
<set name="stockDailyRecords" cascade="delete" table="stock_daily_record" ...>
      <key>
            <column name="STOCK_ID" not-null="true" />
      </key>
      <one-to-many class="com.mkyong.common.StockDailyRecord" />
</set>
```

```
Query q = session.createQuery("from Stock where stockCode = :stockCode ");
q.setParameter("stockCode", "4715");
Stock stock = (Stock)q.list().get(0);
session.delete(stock);
```

Output

```
Hibernate:
    delete from mkyong.stock_daily_record
    where DAILY_RECORD_ID=?

Hibernate:
    delete from mkyong.stock
    where STOCK_ID=?
```

No delete-orphan cascade

You need to delete the 'stockDailyRecords' one by one.

```
StockDailyRecord sdr1 = (StockDailyRecord)session.get(StockDailyRecord.class,
                                            new Integer(56));
StockDailyRecord sdr2 = (StockDailyRecord)session.get(StockDailyRecord.class,
                                            new Integer(57));

session.delete(sdr1);
session.delete(sdr2);
```

Output

```
Hibernate:
    delete from mkyong.stock_daily_record
    where DAILY_RECORD_ID=?
Hibernate:
    delete from mkyong.stock_daily_record
    where DAILY_RECORD_ID=?
```

With delete-orphan cascade

The **cascade="delete-orphan"** is declared in 'stockDailyRecords' to enable the delete orphan cascade effect. When you save or update the Stock, it will remove those 'stockDailyRecords' which already mark as removed.

```
<!-- Stock.hbm.xml -->
<set name="stockDailyRecords" cascade="delete-orphan" table="stock_daily_record" >
      <key>
            <column name="STOCK_ID" not-null="true" />
      </key>
      <one-to-many class="com.mkyong.common.StockDailyRecord" />
</set>
```

```
StockDailyRecord sdr1 = (StockDailyRecord)session.get(StockDailyRecord.class,
                                       new Integer(56));
StockDailyRecord sdr2 = (StockDailyRecord)session.get(StockDailyRecord.class,
                                       new Integer(57));

Stock stock = (Stock)session.get(Stock.class, new Integer(2));
stock.getStockDailyRecords().remove(sdr1);
stock.getStockDailyRecords().remove(sdr2);

session.saveOrUpdate(stock);
```

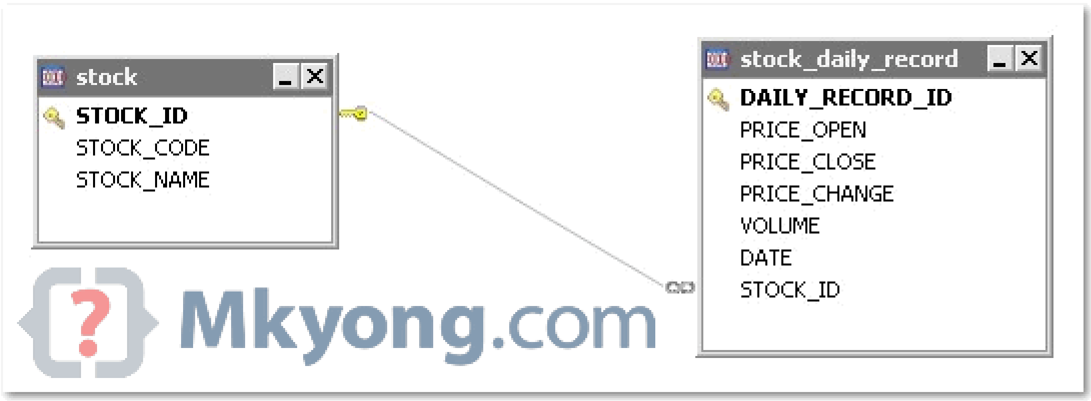
Output

```
Hibernate:
    delete from mkyong.stock_daily_record
    where DAILY_RECORD_ID=?
Hibernate:
    delete from mkyong.stock_daily_record
    where DAILY_RECORD_ID=?
```

In short, delete-orphan allow parent table to delete few records (delete orphan) in its child table.

However, there is no relationship between cascade and inverse, both are totally different notions. The "**inverse**" keyword is always declare in **one-to-many** and **many-to-many** relationship (many-to-one doesn't has inverse keyword), it means which side is responsible to take care of the relationship.

In short, inverse="true" means this is the relationship owner, and inverse="false" (default) means it's not.



1. inverse="true"

If inverse="true" in the set variable, it means "stock_daily_record" is the relationship owner, so Stock will NOT UPDATE the relationship.

```
<class name="com.mkyong.common.Stock" table="stock" ...>
    ...
        <set name="stockDailyRecords" table="stock_daily_record" inverse="true" >
```

2. inverse="false"

If inverse="false" (default) in the set variable, it means "stock" is the relationship owner, and Stock will UPDATE the relationship.

```
<class name="com.mkyong.common.Stock" table="stock" ...>
        ...
        <set name="stockDailyRecords" table="stock_daily_record" inverse="false" >
```

4. inverse="false" Example

If keyword "inverse" is not define, the inverse = "false" will be used, which is

```
<!--Stock.hbm.xml-->
<class name="com.mkyong.common.Stock" table="stock" ...>
        ...
        <set name="stockDailyRecords" table="stock_daily_record" inverse="false">
```

It means "stock" is the relationship owner, and it will maintains the relationship.

24. **What do you mean by Named – SQL query?**

Named SQL queries are defined in the mapping xml document and called wherever required.
Example:

```
<sql-query name = "empdetails">
  <return alias="emp" class="com.test.Employee"/>
    SELECT emp.EMP_ID AS {emp.empid},
           emp.EMP_ADDRESS AS {emp.address},
           emp.EMP_NAME AS {emp.name}
    FROM Employee EMP WHERE emp.NAME LIKE :name
</sql-query>
```

Invoke Named Query :

```
List people = session.getNamedQuery("empdetails")
                    .setString("TomBrady", name)
                    .setMaxResults(50)
                    .list();
```

25. **How do you invoke Stored Procedures?**

```
<sql-query name="selectAllEmployees_SP" callable="true">
 <return alias="emp" class="employee">
   <return-property name="empid" column="EMP_ID"/>

   <return-property name="name" column="EMP_NAME"/>
   <return-property name="address" column="EMP_ADDRESS"/>
    { ? = call selectAllEmployees(dept) }
 </return>
</sql-query>

SQLQuery sq = (SQLQuery) session.getNamedQuery("selectAllEmployees_SP ");

sq.addEntity("dept" "SSNT");

List results = sq.list();
```

Or

```
Query q = session.createSQLQuery(" { call selectAllEmployees (?) }");
q.setInteger(0,"SSNt");  // first parameter, index starts with 0
q.executeUpdate();
```

26. **What is Criteria Query?**

Criteria APIs in the hibernate framework is useful for creating the dynamic query to execute. It is an alternative way to write the queries without using HQL. The queries are generated at runtime and executed on the fly. Application developer need not worry about writing the query in hand.

Criteria criteria = session.createCriteria(Student.class) equivalent to “Select * from Student”

This is also a very convenient approach for functionality like "search" screens where there are variable number of conditions to be placed upon the result set.

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

The retrieval of data itself can be separated into four major categories:

1. Projection

2. Restriction

3. Aggregation

4. Grouping

Projection Example

SELECT NAME, ID FROM PRODUCT

Would become

```
List products =session.createCriteria(Product.class).setProjection(
   Projections.propertyList()
      .add(Projection.property(\"name\"))
      .add(Projection.property(\"id\"))
   ).list();
```

Restriction Example

SELECT * FROM ORDERS WHERE ORDER_ID=’1092’;

Would become

```
   List orders= session.createCriteria(Order.class)
           .add(Restrictions.eq(“orderId”,”1092”)) .list();
```

Restriction.between is used to apply a "between" constraint to the field.

Restriction.eq is used to apply an "equal" constraint to the field.

Restriction.ge is used to apply a "greater than or equal" constraint to the field.

Restriction.gt is used to apply a "greater than" constraint to the field.

Restriction.idEq is used to apply an "equal" constraint to the identifier property.

Restriction.in is used to apply an "in" constraint to the field.

Restriction.isNotNull is used to apply an "is not null" constraint to the field.

Restriction.isNull is used to apply an "is null" constraint to the field.

Restriction.ne is used to apply a "not equal" constraint to the field.

Aggregation Example

```
SELECT O.*, P.* FROM ORDERS O, PRODUCT P WHERE O.ORDER_ID=P.ORDER_ID;
```

Would become

```
List orders = session.createCriteria(Order.class)
          .setFetchMode(“products”,FetchMode.JOIN) .list();
```

Grouping Example

```
SELECT COUNT(ID) FROM ORDER  HAVING PRICETOTAL>2000 GROUP BY ID
```

Can be rewritten in Criteria query as follows:

```
List orders = session.createCriteria(Order.class)
    .setProjection( Projections.projectionList()
     .add( Projections.count(“id”) )
      .add( Projections.groupProperty(“id”) )
    ) .list();
```

27. **Define HibernateTemplate**

org.springframework.orm.hibernate.HibernateTemplate is a helper class which provides different methods for querying/retrieving data from the database. It also converts checked HibernateExceptions into unchecked DataAccessExceptions.

28. **What are the benefits does HibernateTemplate provide?**

The benefits of HibernateTemplate are :

- HibernateTemplate, a Spring Template class simplifies interactions with Hibernate Session.
- Common functions are simplified to single method calls.
- Sessions are automatically closed.
- Exceptions are automatically caught and converted to runtime exceptions.

29.

30. **How do you switch between relational databases without code changes?**

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

31. **If you want to see the Hibernate generated SQL statements on console, what should we do?**

In Hibernate configuration file set as follows:

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

32. **What are derived properties?**

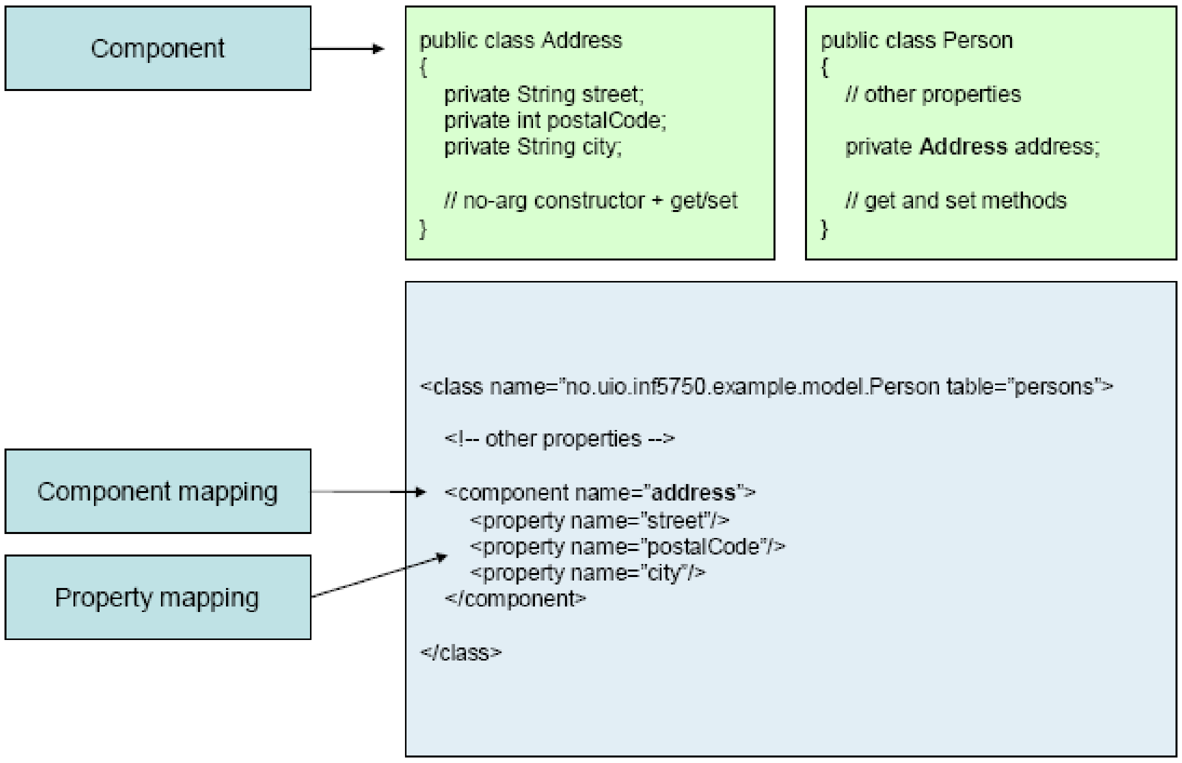
The properties that are actually not mapped to a column, but calculated at runtime by evaluation of an expression are called derived properties. The expression can be defined using the formula attribute of the element. These properties are by definition read-only, the property value is computed at load time. You declare the computation as a SQL expression, this translates to a SELECT clause subquery in the SQL query that loads an instance

```
<property name="totalPrice"
formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p   WHERE
li.productId = p.productId
AND li.customerId = customerId
AND li.orderNumber = orderNumber )"/>
```

33. **What is component mapping in Hibernate?**

- A component is an object saved as a value, not as a reference
- A component can be saved directly without needing to declare interfaces or identifier properties
- Required to define an empty constructor

- Shared references not supported



```
public class Address
{
    private String street;
    private int postalCode;
    private String city;

    // no-arg constructor + get/set
}
```

```
public class Person
{
    // other properties

    private Address address;

    // get and set methods
}
```

```
<class name="no.uio.inf5750.example.model.Person table="persons">

    <!-- other properties -->

    <component name="address">
        <property name="street"/>
        <property name="postalCode"/>
        <property name="city"/>
    </component>

</class>
```

34. **What is the difference between sorted and ordered collection in hibernate?**

sorted collection	**order collection**
A sorted collection is sorting a collection by utilizing the sorting features provided by the Java collections framework. The sorting occurs in the memory of JVM which running Hibernate, after the data being read from database using java comparator.	Order collection is sorting a collection by specifying the order-by clause for sorting this collection when retrieval.
If your collection is not large, it will be more efficient way to sort it.	If your collection is very large, it will be more efficient way to sort it .

35. **What is the advantage of Hibernate over jdbc?**

JDBC	Hibernate
With JDBC, developer has to write code to map an object model's data representation to a relational data model and its corresponding database schema.	Hibernate is flexible and powerful ORM solution to map Java classes to database tables. Hibernate itself takes care of this mapping using XML files so developer does not need to write code for this.
With JDBC, the automatic mapping of Java objects with database tables and vice versa conversion is to be taken care of by the developer manually with lines of code.	Hibernate provides transparent persistence and developer does not need to write code explicitly to map database tables tuples to application objects during interaction with RDBMS.
JDBC supports only native Structured Query Language (SQL). Developer has to find out the efficient way to access database, i.e. to select effective query from a number of queries to perform same task.	Hibernate provides a powerful query language Hibernate Query Language (independent from type of database) that is expressed in a familiar SQL like syntax and includes full support for polymorphic queries. Hibernate also supports native SQL statements. It also selects an effective way to perform a database manipulation task for an application.
Application using JDBC to handle persistent data (database tables) having database specific code in large amount. The code written to map table data to application objects and vice versa is actually to map table fields to object properties. As table changed or database changed then it's essential to change object structure as well as to change code written to map table-to-object/object-to-table.	Hibernate provides this mapping itself. The actual mapping between tables and application objects is done in XML files. If there is change in Database or in any table then the only need to change XML file properties.
With JDBC, it is developer's responsibility to handle JDBC result set and convert it to Java objects through code to use this persistent data in application. So with JDBC, mapping between Java objects and database tables is done manually.	Hibernate reduces lines of code by maintaining object-table mapping itself and returns result to application in form of Java objects. It relieves programmer from manual handling of persistent data, hence reducing the development time and maintenance cost.
With JDBC, caching is maintained by hand-coding.	Hibernate, with Transparent Persistence, cache is set to application work space.

	Relational tuples are moved to this cache as a result of query. It improves performance if client application reads same data many times for same write. Automatic Transparent Persistence allows the developer to concentrate more on business logic rather than this application code.
In JDBC there is no check that always every user has updated data. This check has to be added by the developer.	Hibernate enables developer to define version type field to application, due to this defined field Hibernate updates version field of database table every time relational tuple is updated in form of Java class object to that table. So if two users retrieve same tuple and then modify it and one user save this modified tuple to database, version is automatically updated for this tuple by Hibernate. When other user tries to save updated tuple to database then it does not allow saving it because this user does not have updated data.

36. **What are the Collection types in Hibernate?**

- Bag
- Set
- List
- Array
- Map

37. **What are the ways to express joins in HQL?**

HQL provides four ways of expressing (inner and outer) joins:-

- An *implicit* association join
- An ordinary join in the FROM clause
- A fetch join in the FROM clause.
- A *theta-style* join in the WHERE clause.

38. **What is Hibernate proxy?**

An object proxy is just a way to avoid retrieving an object until you need it. When you actually call load () method on session it returns you proxy. This proxy may contain actual method to load the data. The proxy attribute enables lazy initialization of persistent instances of the class. Hibernate will initially return CGLIB proxies which implement the named interface. The actual persistent object will be loaded when a method of the proxy is invoked.

when we call the load () method as given below
session.load(empObject);
here it is showing that empObject is loaded.But in practicality load() method is lazy loading so it is going to create proxy class and when we call the value in the table using getter method in that instance it will touch the database and give the data form database to session object.i.e empObject.getEname(); when we call this method in that instance it will touch the database and put the data into session object.

39. **How can Hibernate be configured to access an instance variable directly and not through a setter method?**

By mapping the property with **access="field"** in Hibernate metadata. This force hibernates to bypass the setter method and access the instance variable directly while initializing a newly loaded object.

40. **How can a whole class be mapped as immutable?**

Mark the class as mutable="false" (Default is true), This specifies that instances of the class are (not) mutable. Immutable classes may not be updated or deleted by the application.

41. **What is the use of dynamic-insert and dynamic-update attributes in a class mapping**

- dynamic-update (defaults to false): Specifies that UPDATE SQL should be generated at runtime and contain only those columns whose values have changed
- dynamic-insert (defaults to false): Specifies that INSERT SQL should be generated at runtime and contain only the columns whose values are not null.

42. **What do you mean by fetching strategy ?**

A *fetching strategy* is the strategy Hibernate will use for retrieving associated objects if the application needs to navigate the association. Fetch strategies may be declared in the O/R mapping metadata, or over-ridden by a particular HQL or Criteria query.

There are four fetching strategies

1. fetch-"join" = Disable the lazy loading, always load all the collections and entities.
2. fetch-"select" (default) = Lazy load all the collections and entities.

3. batch-size="N" = Fetching up to 'N' collections or entities, *Not record*.
4. fetch-"subselect" = Group its collection into a sub select statement.

```
...
@Entity
@Table(name = "stock", catalog = "mkyong")
public class Stock implements Serializable{
...
        @OneToMany(fetch = FetchType.LAZY, mappedBy = "stock")
        @Cascade(CascadeType.ALL)
        @Fetch(FetchMode.SELECT)
        @BatchSize(size = 10)
        public Set<StockDailyRecord> getStockDailyRecords() {
                return this.stockDailyRecords;
        }
...
}
```

1. fetch="select" or @Fetch(FetchMode.SELECT)

This is the default fetching strategy. it enabled the lazy loading of all it's related collections. Let see the example…

```
//call select from stock
Stock stock = (Stock)session.get(Stock.class, 114);
Set sets = stock.getStockDailyRecords();

//call select from stock_daily_record
for ( Iterator iter = sets.iterator();iter.hasNext(); ) {
      StockDailyRecord sdr = (StockDailyRecord) iter.next();
      System.out.println(sdr.getDailyRecordId());
      System.out.println(sdr.getDate());
}
```

Output

```
Hibernate:
    select ...from mkyong.stock
    where stock0_.STOCK_ID=?

Hibernate:
    select ...from mkyong.stock_daily_record
    where stockdaily0_.STOCK_ID=?
```

Hibernate generated two select statements

1. Select statement to retrieve the Stock records -**session.get(Stock.class, 114)**
2. Select its related collections – **sets.iterator()**

2. fetch="join" or @Fetch(FetchMode.JOIN)

The "join" fetching strategy will disabled the lazy loading of all it's related collections. Let see the example…

```
//call select from stock and stock_daily_record
Stock stock = (Stock)session.get(Stock.class, 114);
Set sets = stock.getStockDailyRecords();

//no extra select
for ( Iterator iter = sets.iterator();iter.hasNext(); ) {
      StockDailyRecord sdr = (StockDailyRecord) iter.next();
      System.out.println(sdr.getDailyRecordId());
      System.out.println(sdr.getDate());
}
```

Output

```
Hibernate:
    select ...
    from
        mkyong.stock stock0_
    left outer join
        mkyong.stock_daily_record stockdaily1_
            on stock0_.STOCK_ID=stockdaily1_.STOCK_ID
    where
        stock0_.STOCK_ID=?
```

Hibernate generated only one select statement, it retrieve all its related collections when the Stock is initialized. -**session.get(Stock.class, 114)**

3. batch-size="10" or @BatchSize(size = 10)

This 'batch size' fetching strategy is always misunderstanding by many Hibernate developers. Let see the *misunderstand* concept here…

```
Stock stock = (Stock)session.get(Stock.class, 114);
Set sets = stock.getStockDailyRecords();

for ( Iterator iter = sets.iterator();iter.hasNext(); ) {
      StockDailyRecord sdr = (StockDailyRecord) iter.next();
      System.out.println(sdr.getDailyRecordId());
      System.out.println(sdr.getDate());
}
```

What is your expected result, is this per-fetch 10 records from collection? See the output
Output

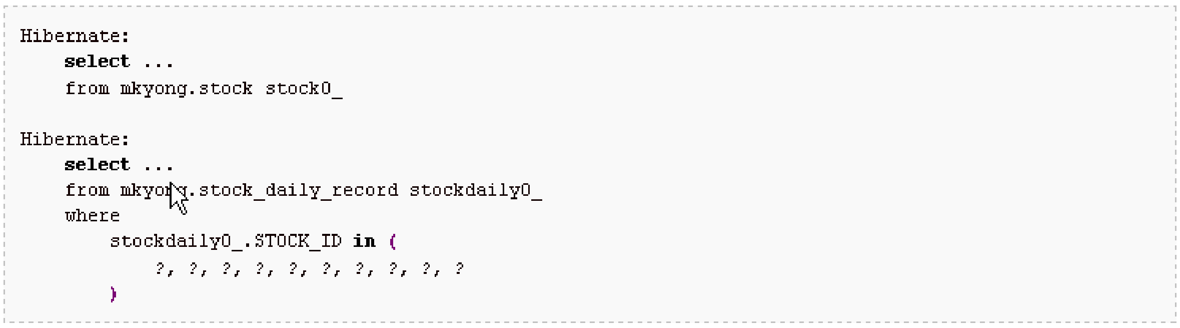
```
Hibernate:
    select ...from mkyong.stock
    where stock0_.STOCK_ID=?

Hibernate:
    select ...from mkyong.stock_daily_record
    where stockdaily0_.STOCK_ID=?
```

The batch-size did nothing here, it is not how batch-size work. See this statement.

The batch-size fetching strategy is not define how many records inside in the collections are loaded. Instead, it defines how many collections should be loaded.

Enabled the batch-size='10' fetching strategy

Let see another example with batch-size='10' is enabled.
Output



```
Hibernate:
    select ...
    from mkyong.stock stock0_

Hibernate:
    select ...
    from mkyong.stock_daily_record stockdaily0_
    where
        stockdaily0_.STOCK_ID in (
            ?, ?, ?, ?, ?, ?, ?, ?, ?, ?
        )
```

Now, Hibernate will per-fetch the collections, with a select *in* statement. If you have 20 stock records, it will generate 3 select statements.

1. Select statement to retrieve all the Stock records.
2. Select In statement to per-fetch its related collections (10 collections a time)
3. Select In statement to per-fetch its related collections (next 10 collections a time)

With batch-size enabled, it simplify the select statements from 21 select statements to 3 select statements.

4. fetch="subselect" or @Fetch(FetchMode.SUBSELECT)

This fetching strategy is enable all its related collection in a sub select statement. Let see the same query again..

```
List<Stock> list = session.createQuery("from Stock").list();

for(Stock stock : list){

    Set sets = stock.getStockDailyRecords();

    for ( Iterator iter = sets.iterator();iter.hasNext(); ) {
            StockDailyRecord sdr = (StockDailyRecord) iter.next();
            System.out.println(sdr.getDailyRecordId());
            System.out.println(sdr.getDate());
    }
}
```

Output

```
Hibernate:
    select ...
    from mkyong.stock stock0_

Hibernate:
    select ...
    from
        mkyong.stock_daily_record stockdaily0_
    where
        stockdaily0_.STOCK_ID in (
            select
                stock0_.STOCK_ID
            from
                mkyong.stock stock0_
        )
```

With "subselect" enabled, it will create two select statements.

1. Select statement to retrieve all the Stock records.
2. Select all its related collections in a sub select query.

43. **What is automatic dirty checking?**

While a session remains open, if a Persistent object is modified, its data is kept synchronized with the database. The data will be synchronized (but not committed) when session.flush() is called. It will be synchronized and committed when the transaction is committed. At the end of a unit of work, it knows which objects have been modified. It then calls update statements on all updated objects. This process of monitoring and updating only objects that have changed is called *automatic dirty checking*.

We can say that automatic dirty checking is a feature that saves us the effort of explicitly asking Hibernate to update the database when we modify the state of an object inside a transaction. An important note here is, Hibernate will compare objects by value, except for Collections, which are compared by identity. For this reason you should return exactly the same collection instance as Hibernate passed to the setter method to prevent unnecessary database updates.

44. **What is transactional write-behind?**

Hibernate (and many other ORM implementations) executes SQL statements asynchronously. An INSERT statement isn't usually executed when the application calls Session.save(); an UPDATE isn't immediately issued when the application calls Item.addBid(). Instead, the SQL statements are usually issued at the end of a transaction. This behavior is called write-behind

Listing 2.3 Updating a message

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();

// 1 is the generated id of the first message
Message message =
        (Message) session.load( Message.class, new Long(1) );
message.setText("Greetings Earthling");
Message nextMessage = new Message("Take me to your leader (please)");
message.setNextMessage( nextMessage );
tx.commit();
session.close();
```

This code calls three SQL statements inside the same transaction:

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
where m.MESSAGE_ID = 1

insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (2, 'Take me to your leader (please)', null)

update MESSAGES
set MESSAGE_TEXT = 'Greetings Earthling', NEXT_MESSAGE_ID = 2
where MESSAGE_ID = 1
```

Hibernate uses a sophisticated algorithm to determine an efficient ordering that avoids database foreign key constraint violations but is still sufficiently predictable to the user. This feature is called transactional write-behind.

45. **What are Callback interfaces?**

Callback interfaces allow the application to receive a notification when something interesting happens to an object—for example, when an object is loaded, saved, or deleted. Hibernate applications don't need to implement these callbacks, but they're useful for implementing certain kinds of generic functionality

46. **What are the types of Hibernate instance states?**

Three types of instance states:

transient
The instance is not, and has never been associated with any persistence context. It has no persistent identity (primary key value).

persistent
The instance is currently associated with a persistence context. It has a persistent identity (primary key value) and, perhaps, a corresponding row in the database. For a particular persistence context, Hibernate guarantees that persistent identity is equivalent to Java identity (in-memory location of the object).

detached
The instance was once associated with persistence context, but that context was closed, or the instance was serialized to another process. It has a persistent identity and, perhaps, a corresponding row in the database. For detached instances, Hibernate makes no guarantees about the relationship between persistent identity and Java identity.

47. **What are the differences between EJB 3.0 & Hibernate**

Hibernate	**EJB 3.0**
Session–Cache or collection of loaded objects relating to a single unit of work	**Persistence Context**-Set of entities that can be managed by a given EntityManager is defined by a persistence unit
XDoclet Annotations used to support Attribute Oriented Programming	**Java 5.0 Annotations** used to support Attribute Oriented Programming
Defines HQL for expressing queries to the database	**Defines EJB QL** for expressing queries

Supports Entity Relationships through mapping files and annotations in JavaDoc	**Support Entity Relationships** through Java 5.0 annotations
Provides a Persistence Manager API exposed via the Session, Query, Criteria, and Transaction API	**Provides and Entity Manager Interface** for managing CRUD operations for an Entity
Provides callback support through lifecycle, interceptor, and validatable interfaces	**Provides callback support** through Entity Listener and Callback methods
Entity Relationships are unidirectional. Bidirectional relationships are implemented by two unidirectional relationships	**Entity Relationships are bidirectional or unidirectional**

48. **What are the types of inheritance models in Hibernate?**

There are three types of inheritance models in Hibernate:

- Table per class hierarchy
- Table per subclass
- Table per concrete class

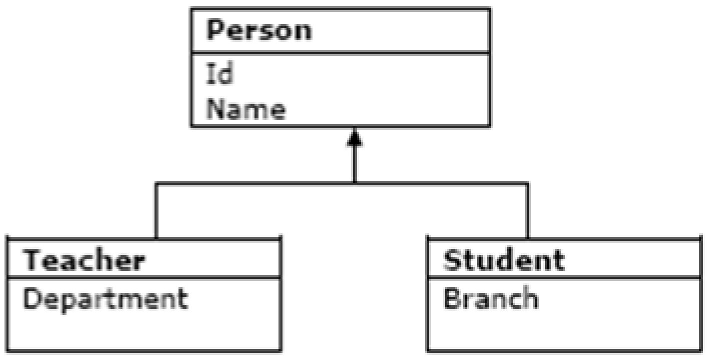
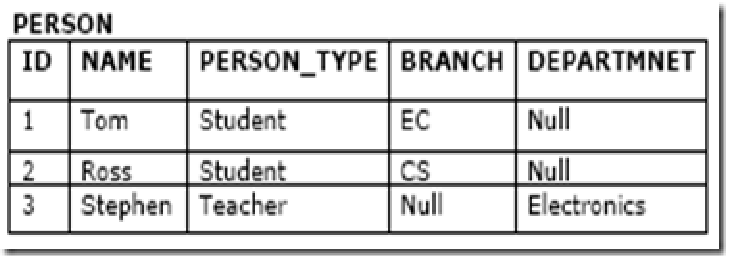


Table per class hierarchy

In this approach the entire hierarchy is mapped to a single table. I.e. All attributes of all the classes are available in a single table as its columns. A discriminator is used to distinguish different classes.



PERSON

ID	NAME	PERSON_TYPE	BRANCH	DEPARTMNET
1	Tom	Student	EC	Null
2	Ross	Student	CS	Null
3	Stephen	Teacher	Null	Electronics

PERSON table consists of all attributes of Person, Teacher and Students as its columns.

Mapping of Table per class hierarchy:

Person.hbm.xml

```
<hibernate-mapping>
    <class name=”Person” table=”PERSON”>
        <id name=”id” column=”ID”>
            <generator class=”increment”/>
        </id>
        <discriminator column=”PERSON_TYPE” type=”string”/>
        <property name= “name” column=”NAME”/>
    </class>
</hibernate-mapping>
```

Discriminator is not a property of any java class, it is a column shared between database and Hibernate. Hibernate using this column to instantiate the appropriate class and populate it accordingly.

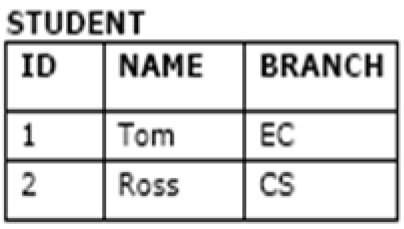
```
Student.hbm.xml
===============

<hibernate-mapping>
    <subclass name=”Student” extends=Person” discriminator-value=”Student”>
        <property name= “branch” column=”BRANCH”/>
    </subclass>
</hibernate-mapping>

Teacher.hbm.xml
===============

<hibernate-mapping>
    <subclass name=”Teacher” extends=Person” discriminator-value=”Teacher”>
        <property name= “department” column=”DEPARTMENT”/>
    </subclass>
</hibernate-mapping>
```

Advantage: This approach is simple and efficient as all data are available in a single table and no joining of tables is required.

Disadvantage: Disadvantage in this mapping strategy is that columns declared by the subclasses cannot have NOT NULL constraints.

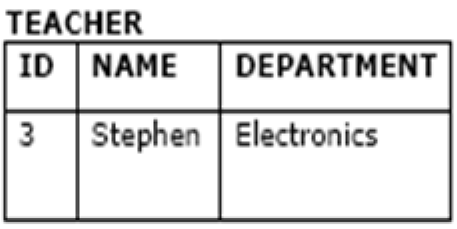
Table per concrete class

In this approach separate class elements are required for each class. Considering the example of Student, Teacher, Person ***we will use two different tables***:



STUDENT

ID	NAME	BRANCH
1	Tom	EC
2	Ross	CS



TEACHER

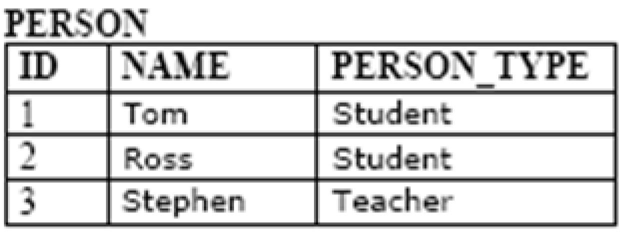
ID	NAME	DEPARTMENT
3	Stephen	Electronics

```
<class name="Person">
    <id name="id" type="long" column="ID">
        <generator class="sequence"/>
    </id>
    <property name="name" column="NAME"/>
    <union-subclass name="Student" table="STUDENT">
        <property name="branch" column="BRANCH"/>
    </union-subclass>
    <union-subclass name="Teacher" table="TEACHER ">
        <property name="branch" column="BRANCH"/>
    </union-subclass>
</class>
```

The limitation of this approach is that if a property is mapped on the super class, the column name must be the same on all subclass tables.

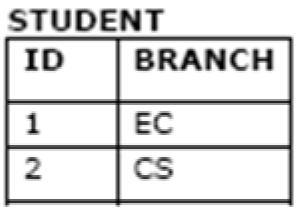
Table per subclass class

In this approach also, separate tables are required for different classes, but will have only specific attributes of subclasses instead of the inherited properties as in table per concrete class.Considering the example of Student, Teacher, Person **we will use three different tables**:



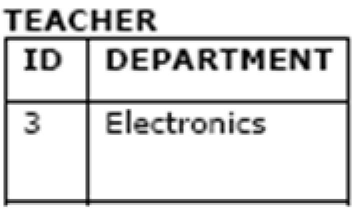
PERSON

ID	NAME	PERSON_TYPE
1	Tom	Student
2	Ross	Student
3	Stephen	Teacher



STUDENT

ID	BRANCH
1	EC
2	CS



TEACHER

ID	DEPARTMENT
3	Electronics

```
<class name="Person" table="PERSON">
    <id name="id" column="ID">
        <generator class="native"/>
    </id>
    <property name="name" column="NAME"/>
    <joined-subclass name="Teacher" table="TEACHER">
        <key column="ID"/>
        <property name="department" column="DEPARTMENT"/>
    </joined-subclass>
    <joined-subclass name="Student" table="STUDENT">
        <key column="ID"/>
        <property name="branch" column="BRANCH"/>
    </joined-subclass>
</class>
```

In this case there are three tables. The subclasses will have primary key with the super class

49. **How to Integrate Struts1.1 , Spring and Hibernate ?**

A: Step 1:

In the struts-config.xml add plugin

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
<set-property property="contextConfigLocation"
value="/WEB-INF/applicationContext.xml"/>
</plug-in>
```

Step 2:
In the applicationContext.xml file

Configure datasourse

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName"><value>oracle.jdbc.driver.OracleDriver</value>
</property>
<property name="url"><value>jdbc:oracle:thin:@10.10.01.24:1541:ebizd</value>
</property>
<property name="username"><value>sa</value></property>
    <property name="password"><value></value></property>
</bean>
```

Step 3.

Configure SessionFactory

```
<!-- Hibernate SessionFactory -->
<bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
<property name="dataSource"><ref local="dataSource"/></property>
<property name="mappingResources">
<list>
<value>com/test/dbxml/User.hbm.xml</value>
</list>
</property>
<property name="hibernateProperties">
<props>
<prop key="hibernate.dialect">net.sf.hibernate.dialect.OracleDialect </prop>
</props>
</property>
</bean>
```

Step 4.

Configure User.hbm.xml

```
<hibernate-mapping>
<class name="com.garnaik.model.User" table="app_user">
<id name="id" column="id" >
<generator class="increment"/>
</id>
<property name="firstName" column="first_name" not-null="true"/>
<property name="lastName" column="last_name" not-null="true"/>
</class>
</hibernate-mapping>
```

Step 5.

In the applicationContext.xml ? configure for DAO

```
<bean id="userDAO" class="com.garnaik.dao.hibernate.UserDAOHibernate">
<property name="sessionFactory"><ref local="sessionFactory"/></property>
</bean>
```

Step 6.

DAO Class

```
public class UserDAOHibernate extends HibernateDaoSupport implements UserDAO {
private static Log log = LogFactory.getLog(UserDAOHibernate.class);
public List getUsers() {
return getHibernateTemplate().find("from User");
}
public User getUser(Long id) {
return (User) getHibernateTemplate().get(User.class, id);
}
public void saveUser(User user) {
getHibernateTemplate().saveOrUpdate(user);
if (log.isDebugEnabled()) {
log.debug("userId set to: " + user.getId());
}
}
public void removeUser(Long id) {
Object user = getHibernateTemplate().load(User.class, id);
getHibernateTemplate().delete(user);
}
}
```

50. **How to prevent concurrent update in Hibernate?**

A: version checking used in hibernate when more then one thread trying to access same data.

For example :

User A edit the row of the TABLE for update (In the User Interface changing data - This is user thinking time) and in the same time User B edit the same record for update and click the update.Then User A click the Update and update done. Change made by user B is gone. In hibernate you can prevent slate object updatation using version checking. Check the version of the row when you are updating the row. Get the version of the row when you are fetching the row of the TABLE for update.On the time of updation just fetch the version number and match with your version number (on the time of fetching).

Steps 1:

Declare a variable "versionId" in your Class with setter and getter.

```
public class Campign {
private Long versionId;
private Long campignId;
private String name;
public Long getVersionId() {
return versionId;
}
public void setVersionId(Long versionId) {
this.versionId = versionId;
}
```

Step 2.

In the .hbm.xml file

```
<class name="beans.Campign" table="CAMPIGN" optimistic-lock="version">
<id name="campignId" type="long" column="cid">
        <generator class="sequence">
                <param name="sequence">CAMPIGN_ID_SEQ</param>
        </generator>
</id>
<version name="versionId" type="long" column="version" />
<property name="name" column="c_name"/>
</class>
```

Step 4.

In the code

```
// foo is an instance loaded by a previous Session
session = sf.openSession();
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() );
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateException();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.close();
```

Hibernate autumatically create/update the version number when you update/insert any row in the table.

51. **What is version checking in Hibernate or How to handle user think time using hibernate ?**

A: version checking used in hibernate when more then one thread trying to access same data. As we already have discussed in the previous example. User A edit the row of the TABLE for update (In the User Interface changing data - This is user thinking time) and in the same time User B edit the same record for update and click the update.

52. **Transaction with plain JDBC in Hibernate?**

If we don't have JTA and don't want to deploy it along with your application, you will usually have to fall back to JDBC transaction demarcation. Instead of calling the JDBC API you better use Hibernate's Transaction and the built-in session-per-request functionality: To enable the thread-bound strategy in your Hibernate configuration:

set hibernate.transaction.factory_class to org.hibernate.transaction.JDBCTransactionFactory
set hibernate.current_session_context_class to thread

```
Session session = factory.openSession();
Transaction tx = null;
try {
tx = session.beginTransaction();
// Do some work
session.load(...);
session.persist(...);
tx.commit(); // Flush happens automatically
}
catch (RuntimeException e) {
tx.rollback();
throw e; // or display error message
}
finally {
session.close();
}
```

53. **What are the general considerations or best practices for defining your Hibernate persistent classes?**

1. You must have a default no-argument constructor for your persistent classes and there should be getXXX() (i.e accessor/getter) and setXXX(i.e. mutator/setter) methods for all your persistable instance variables.
2. You should implement the equals() and hashCode() methods based on your business key and it is important not to use the id field in your equals() and hashCode() definition if the id field is a surrogate key (i.e. Hibernate managed identifier). This is because the Hibernate only generates and sets the field when saving the object.
3. It is recommended to implement the Serializable interface. This is potentially useful if you want to migrate around a multi-processor cluster.
4. The persistent class should not be final because if it is final then lazy loading cannot be used by creating proxy objects.

54. **Difference between session.update() and session.lock() in Hibernate ?**

Both of these methods and saveOrUpdate() method are intended for reattaching a detached object. The session.lock() method simply reattaches the object to the session without checking or updating the database on the assumption that the database in sync with the detached object. It is the best practice to use either session.update(..) or session.saveOrUpdate().

Use session.lock() only if you are absolutely sure that the detached object is in sync with your detached object or if it does not matter because you will be overwriting all the columns that would have changed later on within the same transaction.

Each interaction with the persistent store occurs in a new Session. However, the same persistent instances are reused for each interaction with the database. The application manipulates the state of detached instances originally loaded in another Session and then "reassociates" them using Session.update() or Session.saveOrUpdate().

You may also call lock() instead of update() and use LockMode.READ (performing a version check, bypassing all caches) if you are sure that the object has not been modified.

55. **Difference between getCurrentSession() and openSession() in Hibernate ?**

<u>getCurrentSession() :</u>

The "current session" refers to a Hibernate Session bound by Hibernate behind the scenes, to the transaction scope. A Session is opened when getCurrentSession() is called for the first time and closed when the transaction ends. It is also flushed automatically before the transaction commits. You can call getCurrentSession() as often and anywhere you want as long as the transaction runs.

To enable this strategy in your Hibernate configuration:
set hibernate.transaction.manager_lookup_class to a lookup strategy for your J2EE container
set hibernate.transaction.factory_class to org.hibernate.transaction.JTATransactionFactory

Only the Session that you obtained with sf.getCurrentSession() is flushed and closed automatically.

Example :

```
try {
UserTransaction tx = (UserTransaction)new InitialContext()
.lookup("java:comp/UserTransaction");
tx.begin();
// Do some work
sf.getCurrentSession().createQuery(...);
sf.getCurrentSession().persist(...);
tx.commit();
}
catch (RuntimeException e) {
tx.rollback();
throw e; // or display error message
}
```

openSession()

If you decide to use manage the Session yourself the go for sf.openSession() , you have to flush() and close() it and it does not flush and close() automatically.

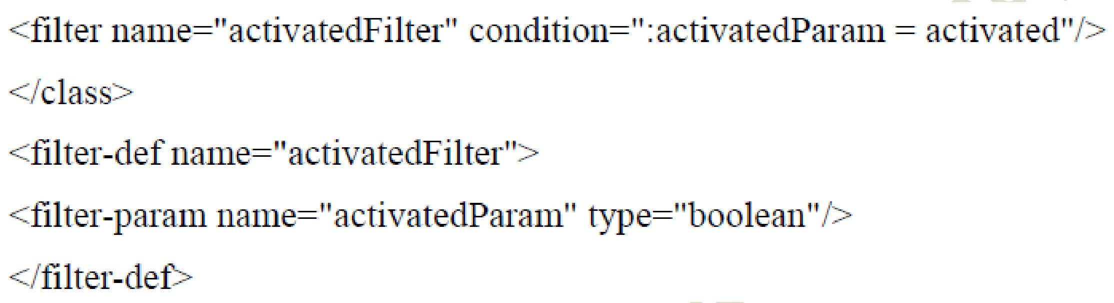
Example :

```
UserTransaction tx = (UserTransaction)new InitialContext()
.lookup("java:comp/UserTransaction");
Session session = factory.openSession();
try {
tx.begin();
// Do some work
session.createQuery(...);
session.persist(...);
session.flush(); // Extra work you need to do
tx.commit();
}
catch (RuntimeException e) {
tx.rollback();
throw e; // or display error message
}
finally {
session.close(); // Extra work you need to do
}
```

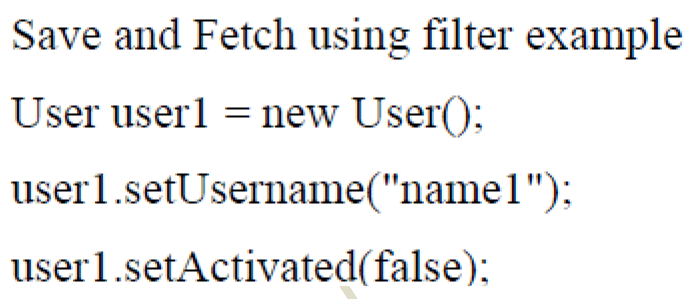
56. **Filter in Hibernate with Example?**

With Hibernate3 there is a new way to filtering the results of searches. Sometimes it is required to only process a subset of the data in the underlying Database tables. Hibernate filters are very useful in those situations. Other approaches for these kind of problems is to use a database view or use a WHERE clause in the query or Hibernate Criteria API. But Hibernate filters can be enabled or disabled during a Hibernate session. Filters can be parameterized also.

'activated' is the property in POJO



```
<filter name="activatedFilter" condition=":activatedParam = activated"/>
</class>
<filter-def name="activatedFilter">
<filter-param name="activatedParam" type="boolean"/>
</filter-def>
```



```
Save and Fetch using filter example
User user1 = new User();
user1.setUsername("name1");
user1.setActivated(false);
```

```
session.save(user1);
User user2 = new User();
user2.setUsername("name2");
user2.setActivated(true);
session.save(user2);
User user3 = new User();
user3.setUsername("name3");
user3.setActivated(true);
session.save(user3);
User user4 = new User();
user4.setUsername("name4");
user4.setActivated(false);
session.save(user4);
```

All the four user saved to Data Base User Table.

Now Fetch the User using Filter..

```
Filter filter = session.enableFilter("activatedFilter");
filter.setParameter("activatedParam",new Boolean(true));
Query query = session.createQuery("from User");
Iterator results = query.iterate();
while (results.hasNext())
{
User user = (User) results.next();
System.out.print(user.getUsername() + " is ");
}
```

Guess the Result :

name2 name3

Because Filer is filtering (only true value) data before query execute.

57. **Criteria Query Two Condition**

```
List organizationList = session.createCriteria(Organization.class)
.add(Restrictions.eq("town","pune"))
.add(Restrictions.eq("statusCode","A"))
    .list();
```

58. **Explain the hibernate N+1 problem and its solution?**

If an application does use the correct fetching strategy to load the data it needs, it may end up making more round trips to the database than necessary.

Lets take an example of the class Contact, which has a one-to-many relationship with Manufacturer (that is, there is one Contact for many Manufacturers) . Since the Contract is uses lazy initialization in its *hbm.xml* file

```
<class name="example.domain.Contact" table="CONTACT" lazy = "true">
...
</class>
```

The execution of the HQL "from Manufacturer manufacturer". This query will not load the data for Contact, but will instead load a proxy to the real data.

The problem is when you run this query but decide in writing your application that you do want to retrieve all the contacts for the returned set of manufacturers

```
Query query = getSupport().getSession().createQuery("from Manufacturer
manufacturer");
List list = query.list();
for (Iterator iter = list.iterator(); iter.hasNext();)
{
  Manufacturer manufacturer = (Manufacturer) iter.next();
  System.out.println(manufacturer.getContact().getName());
}
```

Since the initial query "from Manufacturer manufacturer" does not initialize the Contact instances, an additional separate query is needed to do so for each Contact loaded. you get the N+1 selects problem

Solution
We solve this problem by making sure that the initial query fetches all the data needed to load the objects we need in their appropriately initialized state. One way of doing this is using an HQL fetch join.

We use the HQL

```
"from Manufacturer manufacturer join fetch manufacturer.contact
contact"
```

with the fetch statement. This results in an inner join:

```
select MANUFACTURER.id from manufacturer and contact ... from
MANUFACTURER inner join CONTACT on MANUFACTURER.CONTACT_ID=CONTACT.id
```

Using a *Criteria* query we can get the same result from

```
Criteria criteria = session.createCriteria(Manufacturer.class);
criteria.setFetchMode("contact", FetchMode.EAGER);
```

which creates the SQL

```
select MANUFACTURER.id from MANUFACTURER left outer join CONTACT on
MANUFACTURER.CONTACT_ID=CONTACT.id where 1=1
```

In both cases, our query returns a list of Manufacturer objects with the contact initialized. Only one query needs to be run to return all the contact and manufacturer information required for the example.

59. **Difference between list() and iterate() i9n Hibernate?**

If instances are already be in the session or second-level cache iterate() will give better performance.
If they are not already cached, iterate () will be slower
than list() and might require many database hits for a simple query.

60. **How you can delete persistent objects?**

Session.delete() will remove an object's state from the database. Of course, your application might still hold a reference to a deleted object. It's best to think of delete() as making a persistent instance transient.

session.delete(cat);

61. **SQL statements execution order.**

1. all entity insertions, in the same order the corresponding objects were saved using Session.save()
2. all entity updates
3. all collection deletions
4. all collection element deletions, updates and insertions
5. all collection insertions
6. all entity deletions, in the same order the corresponding

62. **Modifying persistent objects?**

A: DomesticCat cat = (DomesticCat) sess.load(Cat.class, new Long(69));
cat.setName("PK");
sess.flush(); // changes to cat are automatically detected and persisted To Data Base.
No need any session.update() call.

63. **SQL Queries In Hibernate..**

A: You may express a query in SQL, using createSQLQuery() and let Hibernate take care of the mapping from result sets to objects. Note that you may at any time call session.connection() and use the JDBC Connection directly. If you chose to use the Hibernate API, you must enclose SQL aliases in braces:

```
List cats = session.createSQLQuery( "SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10",
"cat", Cat.class ).list();
List cats = session.createSQLQuery( "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
"{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " + "FROM CAT {cat} WHERE
ROWNUM<10", "cat", Cat.class ).list()
```

64. **Equal and Not Equal criteria query.**

List of organisation where town equals to pune.

List organizationList = session.createCriteria(Organization.class).add(Restrictions.eq ("town","pune")).list();

List of organisation where town not equals pune.

List organizationList = session.createCriteria (Organization.class). add(Restrictions.ne ("town","pune")). list();

65. **What is hibernate entity manager?**

Hibernate EntityManager implements:

- The standard Java Persistence management API
- The standard Java Persistence Query Language
- The standard Java Persistence object lifecycle rules
- The standard Java Persistence configuration and packaging

Hibernate EntityManager wraps the powerful and mature Hibernate Core. You can fall back to Hibernate native APIs, native SQL, and native JDBC whenever necessary.

66. **What is Hibernate Annotations?**

Hibernate needs a metadata to govern the transformation of data from POJO to database tables and vice versa. Most commonly XML file is used to write the metadata information in Hibernate. The Java 5 (Tiger) version has introduced a powerful way to provide the metadata to the JVM. The mechanism is known as Annotations. Annotation is the java class which is read through reflection mechanism during the runtime by JVM and does the processing accordingly. The Hibernate Annotations is the powerful way to provide the metadata for the Object and Relational Table mapping. 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. This also reduces the management of different files for the metadata and java code.

Requirements: At a minimum, you need JDK 5.0 and Hibernate Core, but no application server or EJB 3.0 container. You can use Hibernate Core and Hibernate Annotations in any Java EE 5.0 or Java SE 5.0 environment.

67. **One to Many Bi-directional Relation in Hibernate?**

In Parent Table

```
<set name="children">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

In Child Table

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

Example

PROCESS_TYPE_LOV (PROCESS_TYPE_ID number, PROCESS_TYPE_NAME varchar)
PROCESS (PROCESS_ID number, PROCESS_NAME varchar, PROCESS_TYPE_ID number)

<u>In ProcessType Bean</u>

Set processes=new HashSet();

```
<Set name="processes" inverse="true" cascade="delete" lazy="false">
        <key column="PROCESS_TYPE_ID" />
        <one-to-many class="com.bean.ProcessBean" />
</set>
```

<u>In ProcessBean</u>

Private ProcessType processType;

```
<many-to-one name="processType" column="PROCESS_TYPE_ID" lazy="false" />
```

68. **One To One Relation In Hibernate ?**

```
<many-to-one name="address"
    column="addressId"
    unique="true"
    not-null="true"/>
```

```
<one-to-one name="person"
    property-ref="address"/>
```

69. **Many To Many Relation In Hibernate ?**

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null,
create table Address ( addressId bigint not null primary key )
```

Person

```
<set name="addresses" table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
        class="Address"/>
</set>
```

Adress

```
<set name="people" inverse="true" table="PersonAddress">
    <key column="addressId"/>
    <many-to-many column="personId"
        class="Person"/>
</set>
```

70. **What does session.refresh() do ?**

A: It is possible to re-load an object and all its collections at any time, using the refresh() method. This is useful when database triggers are used to initialize some of the properties of the object.
For Example - Triger on cat_name coulmn. Trigger is updating hit_count coulmn in the same Cat Table. When Insert data into Cat TABLE trigger update hit_count coulmn to 1. sess.refresh() reload all the data. No need again to select call.

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

71. **How to get JDBC connections in hibernate?**

A: User Session.connection() method to get JDBC Connection.

72. **How will you configure Hibernate?**

A: Step 1> Put Hibernate properties in the classpath.
Step 2> Put .hbm.xml in class path ?

Main Code –

```
Properties propHibernate = new Properties();
propHibernate.load(new FileInputStream(path_properties));
Configuration configuration = new Configuration();
configuration.addFile(path_mapping);
configuration.setProperties(propHibernate);
sessionFactory = configuration.buildSessionFactory();
```

73. **What is a Hibernate Session? Can you share a session object between different theads?**

Session is a light weight and a non-threadsafe object (No, you cannot share it between threads) that represents a single unit-of-work with the database. Sessions are opened by a SessionFactory and then are closed when all work is complete. Session is the primary interface for the persistence service. A session obtains a database connection lazily (i.e. only when required). To avoid creating too many sessions ThreadLocal class can be used as shown below to get the current session no matter how many times you make call to the currentSession() method.

```
public class HibernateUtil {
public static final ThreadLocal local = new ThreadLocal();
public static Session currentSession() throws HibernateException {
Session session = (Session) local.get();
//open a new session if this thread has no session
if(session == null) {
session = sessionFactory.openSession();
local.set(session);
}
return session;
}
}
```

74. **What is the use of addScalar() method in hibernate?**

If you want to make sure that a particular property should be in a specific type then we can use this method which will ensure about that type and not need to check explicitly.

```
Double max = (Double) sess.createSQLQuery("select max(cat.weight) as maxWeight from
cats cat")
.addScalar("maxWeight", Hibernate.DOUBLE);
.uniqueResult();
```

addScalar() method confim that maxWeight is always double type.This way you don't need to check for it is double or not.

75. **What is the main difference between Entity Beans and Hibernate ?**

A: 1)In Entity Bean at a time we can interact with only one data Base. Where as in Hibernate we can
able to establishes the connections to more than One Data Base. Only thing we need to write one more
configuration file.
2) EJB need container like Weblogic, WebSphere but hibernate don't need. It can be run on tomcat.
3) Entity Beans does not support OOPS concepts where as Hibernate does.
4) Hibernate supports multi level cacheing, where as Entity Beans doesn't.
5) In Hibernate C3P0 can be used as a connection pool.
6) Hibernate is container independent. EJB not.

76. **how to create primary key using hibernate?**

```
<id name="userId" column="USER_ID" type="int">
<generator class="increment"/>
</id>
```

77. **Locking In Hibernate?**

Hibernate is having two types of locking: Optimistic and Pessimistic

Optimistic – If you are using versioning or timestamp in your application then by default hibernate will use the locking. Hibernate will always check the version number before updating the persistence object.

Pessimistic – Explicitly we can implement the locking in hibernate by using the API. Locking of rows using SELECT FOR UPDATE syntax. We have to set the lock while we are loading the persistence object from the db.

78. **What is Flushing?**

The call to tx.commit ()synchronizes the Session state with the database. Hibernate then commits the underlying transaction if and only if beginTransaction() started a new transaction (in both managed and non-managed cases). If begin-Transaction() did not start an underlying database transaction, commit() only synchronizes the Session state with the database; it's left to the responsible party (the code that started the transaction in the first place) to end the transaction. This is consistent with the behavior defined by JTA.We've noted that the call to commit () synchronizes the Session state with the database. This is called flushing, a process you automatically trigger when you use the Hibernate Transaction API.

79. **What is Transparent Write Behind?**

When an object is modified it is not immediately propagated to the database , this is called transparent write behind. For example, if a single property of an object is changed twice in the same Transaction, Hibernate only needs to execute one SQL UPDATE.

Another example of the usefulness of transparent write behind is that Hibernate can take advantage of the JDBC batch API when executing multiple UPDATE, INSERT, or DELETE statements. Hibernate flushes occur only at the following times:

- When a Transaction is committed
- Sometimes before a query is executed
- When the application calls Session.flush() explicitly

80. **What is Transaction Demarcation?**

A database transaction groups data-access operations. A transaction is guaranteed to end in one of two ways: it's either committed or rolled back. Hence, database transactions are always truly atomic. Mark the boundaries of the unit of work. You must start the transaction and, at some point, commit the changes. If an error occurs (either while executing operations or when committing the changes), you have to roll back the transaction to leave the data in a consistent state. This is known as transaction demarcation, and (depending on the API you use) it involves more or less manual intervention.

81. **What is Managed/Non-Managed Transactions?**

In a non-managed environment, the JDBC API is used to mark transaction boundaries. You begin a transaction by calling setAutoCommit(false) on a JDBC connection and end it by calling commit(). You may, at any time, force an immediate rollback by calling rollback().If a database connection is in auto commit mode, the database transaction will be committed immediately after each SQL statement, and a new transaction will be started. This can be useful for ad hoc database queries and ad hoc data updates.

In a managed environment, JTA is used not only for distributed transactions, but also for declarative container managed transactions (CMT). CMT allows you to avoid explicit transaction demarcation calls in your application source code; rather, transaction demarcation is controlled by a deployment-specific descriptor. This descriptor defines how a transaction context propagates when a single thread passes through several different EJBs

82. **How do you set the Flush Mode?**

We can control this behaviour by explicitly setting the Hibernate FlushMode via a call to session.setFlushMode(). The flush modes are as follows:

FlushMode.AUTO— the default. Enables the behaviour just described.

FlushMode.COMMIT—it will be flushed only at the end of the database transaction).

FlushMode.NEVER—Lets you specify that only explicit calls to flush () result in synchronization of session state with the database.

83. **What is Transaction Isolation ?**

A particular transaction shouldn't be visible to and shouldn't influence other concurrently running transactions. Several different strategies are used to implement this behaviour, which is called *isolation*

84. **What are the different Isolation issues?**

Lost update—Two transactions both update a row and then the second transaction aborts, causing both changes to be lost. This occurs in systems that don't implement any locking. The concurrent transactions aren't isolated.

Dirty read—One transaction reads changes made by another transaction that hasn't yet been committed. This is very dangerous, because those changes might later be rolled back.

Unrepeatable read—A transaction reads a row twice and reads different state each time. For example, another transaction may have written to the row, and committed, between the two reads.

Second lost updates problem—A special case of an unrepeatable read. Imagine that two concurrent transactions both read a row, one writes to it and commits, and then the second writes to it and commits. The changes made by the first writer are lost.

Phantom read—A transaction executes a query twice, and the second result set includes rows that weren't visible in the first result set. (It need not necessarily be exactly the same query.) This situation is caused by another transaction inserting new rows between the executions of the two queries.

85. **What are different Isolation levels?**

Read uncommitted—Permits dirty reads but not lost updates. One transaction may not write to a row if another uncommitted transaction has already written to it. Any transaction may read any row, however

Read committed - Permits unrepeatable reads but not dirty reads. This may be achieved using momentary shared read locks and exclusive write locks. Reading transactions don't block other transactions from accessing a row. However, an uncommitted writing transaction blocks all other transactions from accessing the row.

Repeatable read —Permits neither unrepeatable reads nor dirty reads. Phantom reads may occur. This may be achieved using shared read locks and exclusive write locks. Reading transactions block writing transactions (but not other reading transactions), and writing transactions block all other transactions.

Serializable—Provides the strictest transaction isolation. It emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. Serializability may not be implemented using only row-level locks; there must be another mechanism that prevents a newly inserted row from becoming visible to a transaction that has already executed a query that would return the row.

1—Read uncommitted isolation

2—Read committed isolation

4—Repeatable read isolation

8—Serializable isolation

Example - hibernate.connection.isolation = 4

86. **What is Locking?**

Locking is a mechanism that prevents concurrent access to a particular item of data. When one transaction holds a lock on an item, no concurrent transaction can read and/or modify this item. A lock might be just a momentary lock, held while the item is being read, or it might be held until the completion of the transaction. A pessimistic lock is a lock that is acquired when an item of data is read and that is held until transaction completion.

The Hibernate LockMode class lets you request a pessimistic lock on a particular item. In addition, you can use the LockMode to force Hibernate to bypass the cache layer or to execute a simple version check

```
Transaction tx = session.beginTransaction();
Category cat = (Category) session.get(Category.class, catId);
cat.setName("New Name");
tx.commit();
```

then you can obtain a pessimistic lock as follows:

```
Transaction tx = session.beginTransaction();
Category cat =
    (Category) session.get(Category.class, catId, LockMode.UPGRADE);
cat.setName("New Name");
tx.commit();
```

With this mode, Hibernate will load the Category using a SELECT...FOR UPDATE, thus locking the retrieved rows in the database until they're released when the transaction ends

87. **What are the different LockModes available in Hibernate**

There are five lock modes provided by hibernate.

- LockMode.NONE—Don't go to the database unless the object isn't in either cache.
- LockMode.READ—Bypass both levels of the cache, and perform a version check to verify that the object in memory is the same version that currently exists in the database.
- LockMode.UPDGRADE—Bypass both levels of the cache, do a version check (if applicable), and obtain a database-level pessimistic upgrade lock, if that is supported.
- LockMode.UPDGRADE_NOWAIT—The same as UPGRADE, but use a SELECT...FOR UPDATE NOWAIT on Oracle. This disables waiting for concurrent lock releases, thus throwing a locking exception immediately if the lock can't be obtained.
- LockMode.WRITE—obtained automatically when Hibernate has written to a row in the current transaction (this is an internal mode; you can't specify it explicitly).

By specifying an explicit LockMode other than LockMode.NONE, you force Hibernate to bypass both levels of the cache and go all the way to the database. We think that most of the time caching is more useful than pessimistic locking, so we don't use an explicit LockMode unless we really need it. Our advice is that if you have a professional DBA on your project, let the DBA decide which transactions require pessimistic locking once the application is up and running

88. **What are different types of caching?**

A cache keeps a representation of current database state close to the application, either in memory or on disk of the application server machine. The cache is a local copy of the data. The cache sits between your application and the database. The cache may be used to avoid a database hit whenever

The application performs a lookup by identifier (primary key) , and the persistence layer resolves an association lazily

There are three main types of cache

Transaction scope—Attached to the current unit of work, which may be an actual database transaction or an application transaction. It's valid and used as long as the unit of work runs. Every unit of work has its own cache.

Process scope—Shared among many (possibly concurrent) units of work or transactions. This means that data in the process scope cache is accessed by concurrently running transactions, obviously with implications on transaction isolation. A process scope cache might store the persistent instances themselves in the cache, or it might store just their persistent state in a disassembled format.

Cluster scope—Shared among multiple processes on the same machine or among multiple machines in a cluster. It requires some kind of *remote process communication* to maintain consistency. Caching information has to be replicated to all nodes in the cluster. For many (not all) applications, cluster scope caching is of dubious value, since reading and updating the cache might be only marginally faster than going straight to the database.

89. **What is Cache Miss ?**

A cache lookup for an item that isn't contained in the cache.

90. **Define Hibernate Cache Architecture?**

Hibernate has a two-level cache architecture

The first-level cache is the Session itself. A session lifespan corresponds to either a database transaction or an application transaction (as explained earlier in this chapter). We consider the cache associated with the Session to be a transaction scope cache. The first-level cache is mandatory and can't be turned off; it also guarantees object identity inside a transaction. The session cache ensures that when the application requests the same persistent object twice in a particular session, it gets back the same (identical) Java instance. This sometimes helps avoid unnecessary database traffic

Note - Whenever you pass an object to save(), update(), or saveOrUpdate(), and whenever you retrieve an object using load(), find(), list(), iterate(), or filter(), that object is added to the session cache. When flush() is subsequently called, the state of that object will be synchronized with the database.

The second-level cache in Hibernate is pluggable and might be scoped to the process or cluster. Use of the second-level cache is optional and can be configured on a per-class and per-association basis. Hibernate also implements a cache for query result sets that integrates closely with the second-level cache.

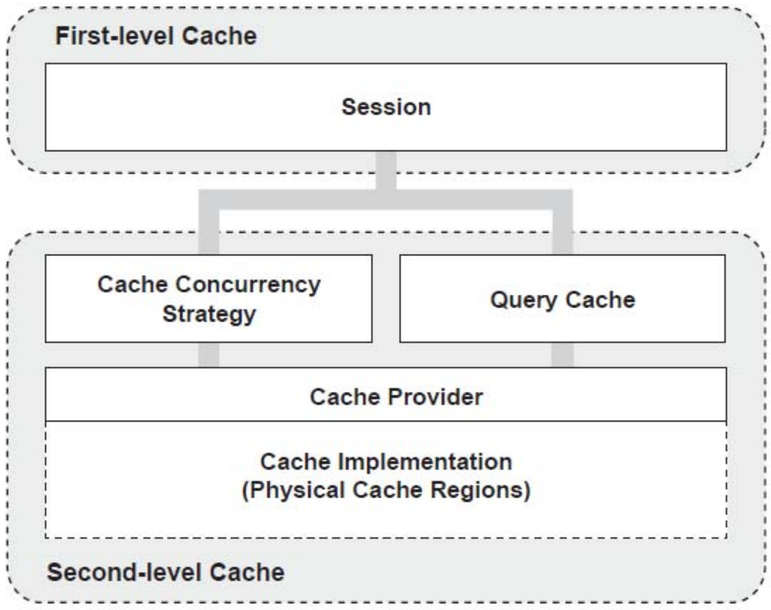

First-level Cache
Session
Cache Concurrency Strategy
Query Cache
Cache Provider
Cache Implementation (Physical Cache Regions)
Second-level Cache

Figure 5.5
Hibernate's two-level cache architecture

The Hibernate second-level cache has process or cluster scope; all sessions share the same second-level cache. The second-level cache actually has the scope of a SessionFactory.

91. **What are the points needs to consider while enabling the cache?**

The cache policy involves setting the following:

■ whether the second-level cache is enabled

■ Hibernate concurrency strategy

■ cache expiration policies (such as timeout, LRU, memory-sensitive)

■ physical format of the cache (memory, indexed files, cluster-replicated)

Note - To repeat, the cache is usually useful only for read mostly classes. If you have data that is updated more often than it's read, don't enable the second-level cache, even if all other conditions for caching are true!

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*

92. **What are the different concurrency strategies available in Hibernate?**

transactional—Available in a managed environment only. It guarantees full transactional isolation up to *repeatable read*, if required. Use this strategy for read-mostly data where it's critical to prevent stale data in concurrent transactions, in the rare case of an update.

read-write—Maintains *read committed* isolation, using a time stamping mechanism. It's available only in non-clustered environments. Again, use this strategy for read-mostly data where it's critical to prevent stale data in concurrent transactions, in the rare case of an update.

nonstrict-read-write—Makes no guarantee of consistency between the cache and the database. If there is a possibility of concurrent access to the same entity, you should configure a sufficiently short expiry timeout. Otherwise, you may read stale data in the cache. Use this strategy if data rarely changes

read-only—A concurrency strategy suitable for data which never changes.Use it for reference data only.

<u>**Note**</u> - With decreasing strictness comes increasing performance. First benchmark your application with the second-level cache disabled. Then enable it for good candidate classes, one at a time, while continuously testing the performance of your system and evaluating concurrency strategies.

93. **What is the use of evict () method?**

If you don't want this synchronization to occur between your persistent object and database, or if you're processing a huge number of objects and need to manage memory efficiently, you can use the evict () method of the Session to remove the object and its collections from the first-level cache. There are several scenarios where this can be useful.

94. **I got an OutOfMemoryException when I try to load 100,000 objects and manipulate all of them. How can I do mass updates with Hibernate?"**

If you insist on using Hibernate even for mass operations, you can immediately evict() each object after it has been processed (while iterating through a query result), and thus prevent memory exhaustion.To completely evict all objects from the session cache, call Session.clear(). We aren't trying to convince you that evicting objects from the first-level cache is a bad thing in general, but that good use cases are rare.

95. **What are different cache providers available in Hibernate?**

EHCache is intended for a simple process scope cache in a single JVM. It can cache in memory or on disk, and it supports the optional Hibernate query result cache.

OpenSymphony OSCache is a library that supports caching to memory and disk in a single JVM, with a rich set of expiration policies and query cache support.

SwarmCache is a cluster cache based on JGroups. It uses clustered invalidation but doesn't support the Hibernate query cache.

JBossCache is a fully transactional replicated clustered cache also based on the JGroups multicast library. The Hibernate query cache is supported, assuming that clocks are synchronized in the cluster.

96. **How will you configure the cache provider in your application?**

<u>Step 1</u> - Look at the mapping files for your persistent classes and decide which cache concurrency strategy you'd like to use for each class and each association.

```
<class
        name="Category"
        table="CATEGORY">
        <cache usage="read-write"/>

        <id ....
    </class>
```

The usage="read-write" attribute tells Hibernate to use a read-write concurrency strategy for the Category cache. Hibernate will now try the second-level cache whenever we navigate to a Category or when we load a Category by identifier.

<u>Step 2</u> - Set the cache provider

hibernate.cache.provider_class = net.sf.ehcache.hibernate.Provider

For example, if you're using OSCache, you should edit oscache.properties, or for

EHCache, ehcache.xml in your classpath.

97. **How do you setup the local cache provider?**

This details will need to be configured in ecache.xml

```
<cache name="org.hibernate.auction.model.Bid"
        maxElementsInMemory="5000"
        eternal="false"
        timeToIdleSeconds="1800"
        timeToLiveSeconds="100000"
        overflowToDisk="false"
    />
```

The timeToIdleSeconds attribute defines the expiry time in seconds since an element was last accessed in the cache. We must set a sensible value here, since we don't want unused bids to consume memory. The timeToLiveSeconds attribute defines the maximum expiry time in seconds since the element was added to the cache. Since bids are immutable, we don't need them to be removed from the cache if they're being accessed regularly. Hence, timeToLiveSeconds is set to a high number. The result is that cached bids are removed from the cache if they have not been used in the past 30 minutes or if they're the least recently used item when the total size of the cache has reached its maximum limit of 5000 elements. We've disabled the disk-based cache in this example, since we anticipate that the application server will be deployed to the same machine as the database. If the expected physical architecture were different, we might enable the disk based cache.

98. **What are the Extension interfaces that are there in hibernate?**

There are many extension interfaces provided by hibernate.
ProxyFactory interface - used to create proxies
ConnectionProvider interface – used for JDBC connection management
TransactionFactory interface – Used for transaction management
Transaction interface – Used for transaction management
TransactionManagementLookup interface – Used in transaction management.
Cahce interface – provides caching techniques and strategies
CacheProvider interface – same as Cache interface
ClassPersister interface – provides ORM strategies
IdentifierGenerator interface – used for primary key generation
Dialect abstract class – provides SQL support

99. **What are different environments to configure hibernate?**

There are mainly two types of environments in which the configuration of hibernate application differs.

i. Managed environment – In this kind of environment everything from database connections, transaction boundaries, security levels and all are defined. An example of this kind of environment is environment provided by application servers such as JBoss, Weblogic and WebSphere.

ii. Non-managed environment – This kind of environment provides a basic configuration template. Tomcat is one of the best examples that provide this kind of environment.

100. **What is the file extension you use for hibernate mapping file?**

The name of the file should be like this : filename.hbm.xml The filename varies here. The extension of these files should be ".hbm.xml". This is just a convention and it's not mandatory. But this is the best practice to follow this extension.

101. **What is meant by Method chaining?**

Method chaining is a programming technique that is supported by many hibernate interfaces. This is less readable when compared to actual java code. And it is not mandatory to use this format. Look how a SessionFactory is created when we use method chaining.

```
SessionFactory sessions = new Configuration()

.addResource("myinstance/MyConfig.hbm.xml")

.setProperties( System.getProperties() )

.buildSessionFactory();
```

102. **What should SessionFactory be placed so that it can be easily accessed?**

As far as it is compared to J2EE environment, if the SessionFactory is placed in JNDI then it can be easily accessed and shared between different threads and various components that are hibernate aware. You can set the SessionFactory to a JNDI by configuring a property hibernate.session_factory_name in the hibernate.properties file.

103. **What is Attribute Oriented Programming?**

XDoclet has brought the concept of attribute-oriented programming to Java. Until JDK 1.5, the Java language had no support for annotations; now XDoclet uses the Javadoc tag format (@attribute) to specify class-, field-, or method-level metadata attributes. These attributes are used to generate hibernate mapping file automatically when the application is built. This kind of programming that works on attributes is called as Attribute Oriented Programming.

104. **What are the different methods of identifying an object?**

There are three methods by which an object can be identified.
i. Object identity –Objects are identical if they reside in the same memory location in the JVM. This can be checked by using the = = operator.
ii. Object equality – Objects are equal if they have the same value, as defined by the equals( ) method. Classes that don't explicitly override this method inherit the implementation defined by java.lang.Object, which compares object identity.
iii. Database identity – Objects stored in a relational database are identical if they represent the same row or, equivalently, share the same table and primary key value.

105. **How can you make a property be read from the database but not modified in anyway (make it immutable)?**

By using the insert="false" and update="false" attributes

106. **How can a whole class be mapped as immutable?**

By using the mutable="false" attribute in the class mapping.

107. **How can I count the number of query results without actually returning them?**

Integer count = (Integer) session.createQuery("select count(*) from ....").uniqueResult();

108. **How can I find the size of a collection without initializing it?**

Integer size = (Integer) s.createFilter(collection, "select count(*)").uniqueResult();

Are collections pageable?

Query q = s.createFilter(collection, ""); // the trivial filter

q.setMaxResults(PAGE_SIZE);

q.setFirstResult(PAGE_SIZE * pageNumber);

List page = q.list();

109. **How can I execute arbitrary SQL using Hibernate?**

PreparedStatement ps = session.connection().prepareStatement(sqlString);