Hibernate Questions - Answers

Disadvantages of jdbc

1. Checked Exception Code-

- we must need to handle the exception in jdbc.
- without try,catch or throws we can't develop single line of code using jdbc.should be used try catch or throws.

2 .Memory Leakage Problems -

- java programmer is responsible to opening and closing the connection if you are not closing the connection then we are getting memory leakage problem.
- our application running inside the server.so many applications running.In our application we are not closing the connection still server is running whenever server is running our application running so still our application pointing to the database and utilizing resources then performance decreases.

3. we need to implementing boiler plate code

- need to write some duplicate code each and every time.
- for example - load driver,username,password,statement,opening connection.

4. Hard to implement MVC concept.
5.doesn't support oop concepts.
6.ORM not supported.
7. Caching,Mapping,Inheritance not supported by JDBC.

To overcome this problem industry peoples introduced concept i.e Entity Bean which is developed by EJB technology.

EJB disadvantages

1. Heavyweight component -
    - should need to implement predefined API's without predefined API's we can't developed project.
2. Testing is more complex
    - to test the component developer need one of the application server i.e weblogic,glassfish,jboss,jrun,IBMSphere etc.
    - webservers are not supported by EJB i.e tomcat,smtp,ftp etc.

3. More number of xml file.

so to overcome this developer introduced new framework.

1.**What is ORM framework? List out some names**

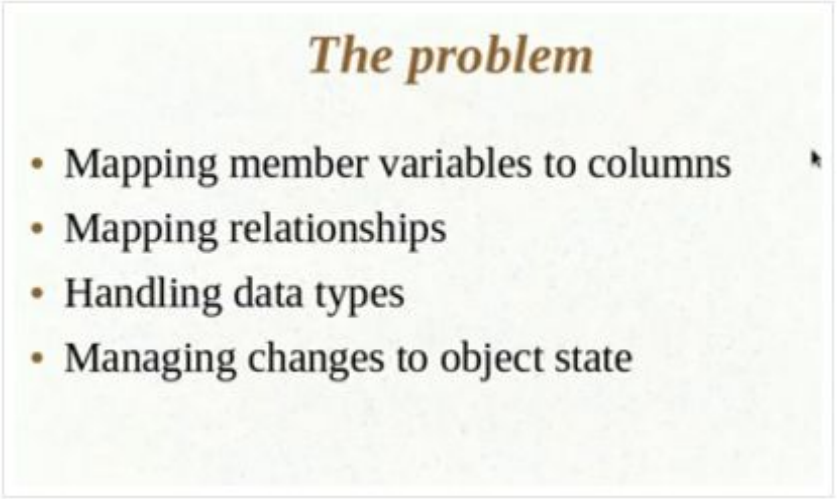
- ORM is an object relational mapping which maps java object into the table of relational database.
- used to persist data objects into relational database.

- List of ORM frameworks :
  1.Hibernate
  2.iBATIS
  3.EJB(Enterprise Java Beans)
  4.JDO(Java Data Objects)
  5.TopLink
  6.JPA(Java Persistence API)

2.**Why to use ORM framework**

- one problem to object save into relational database each entity of the object saved individually to the database table in the associated columns.

- It is more times to taken and complex in case of lots of user objects to be save into the database table.



What is Hibernate?

- Hibernate is lightweight and opensource framework.
- Hibernate supports ORM & OOP concepts.
- testing is very simple.no need of any server.
- database independent.
- Hibernate is non-Invasive Framework

Invasive -
force to developer to extends or implements classes & interfaces.
for example - struts
Non-invasive -
doesn't force to programmer to extends or implements classes & interfaces.
for example - hibernate,spring

- no need to open & close connection.
- no need to put try catch.
- framework convert checked exceptions into unchecked exception.
- hibernate take care of database specific queries no need to write explicitly.
- automatically creates table.
- collections support
- no need to generate sql queries.

1.Hibernate Core interfaces and classes 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
- Transaction interface
- SessionFactory interface
- Session interface
- Query and Criteria interfaces

1. Configuration:

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

➔ Configuration cfg = new Configuration();
cfg.addResource("hello/Message.hbm.xml");
cfg.setProperties(System.getProperties());

2. Transaction:

➔ A transaction represents a unit of work. Application uses transactions to do some operations on DB.

➔ Within one transaction you can do several operations and can commit transaction once after successfully completed all operations.

➔ The advantage here is you can rollback all previous operations if one operation is fail in your operation batch.

➔ The Transaction does not get committed when session gets flushed.

➔ The Transaction interface is an optional API.

➔ Hibernate applications may choose not to use this interface, instead

managing transactions in their own infrastructure code.

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

3. SessionFactory:

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

4. Session:

- ➔ The Session is a persistence manager that manages operation like storing and retrieving objects. Instances of Session are inexpensive to create and destroy. They are not thread safe.
- ➔ Session holds a mandatory first-level cache of persistent objects that are used when navigating the object graph or looking up objects by identifier.
- ➔
- ➔ Session session = sessionFactory.openSession();

5. Query and Criteria interfaces:

- ➔ The Query interface allows you to perform queries against the database and control how the query is executed. Queries are written in HQL or in the native SQL dialect of your database. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.
- ➔ The Criteria interface is very similar; it allows you to create and execute object-oriented criteria queries.
- ➔ To help make application code less verbose, Hibernate provides some shortcut methods on the Session interface that let you invoke a query in one line of code. We won't use these shortcuts; instead, we'll always use the Query interface.
- ➔ A Query instance is lightweight and can't be used outside the Session that created it.

flow of Hibernate communication with RDBMS

General flow of hibernate with RDBMS

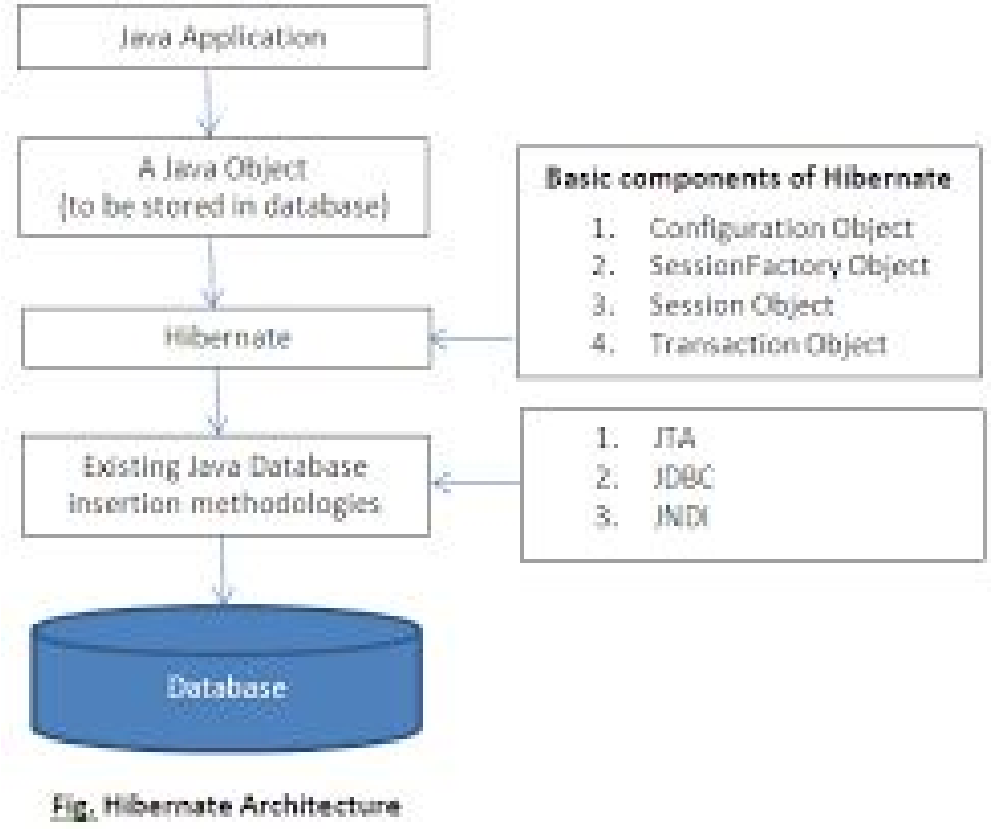


Fig. Hibernate Architecture

2.how to get session factory, session transaction in hibernate..write down code

```
static SessionFactory sessionFactory = null;

   static {
      try{
          sessionFactory = new Configuration().configure().buildSessionFactory();
      }catch (Throwable ex) {
        System.err.println("Failed to create sessionFactory object." + ex);
        throw new ExceptionInInitializerError(ex);
      }
   }
```

session.flush() vs trascation.commit()

- ➔ **flush()** will synchronize your database with the current state of object/objects held in the memory but it does not commit the transaction.
- ➔ So, if you get any exception after flush() is called, then the transaction will be rolled back.

- ➔ You can synchronize your database with small chunks of data using flush() instead of committing a large data at once using commit() and face the risk of getting an **Out Of Memory Exception**.
- ➔ **commit()** will make data stored in the database permanent. There is no way you can rollback your transaction once the commit() succeeds.

Benefits of hibernate over JDBC?

1)Hibernate is database independent, same code will work for all databases like ORACLE,MySQL ,SQLServer etc. In case of JDBC query must be database specific.

2)As Hibernate is set of Objects , you don't need to learn SQL language.
You can treat TABLE as a Object . In case of JDBC you need to learn SQL.

3)Don't need Query tuning in case of Hibernate. If you use Criteria Queries in Hibernate, then hibernate automatically tuned your query and return best result with performance. In case of JDBC you need to tune your queries.

4) You will get benefit of Cache. Hibernate support two level of cache. First level and 2nd level. So you can store your data into Cache for better performance. In case of JDBC you need to implement your java cache .

5)Hibernate supports Query cache and It will provide the statistics about your query and database status. JDBC Not provides any statistics.

6)Development fast in case of Hibernate because you don't need to write queries.

7)No need to create any connection pool in case of Hibernate. You can use c3p0. In case of JDBC you need to write your own connection pool.

8)In the xml file you can see all the relations between tables in case of Hibernate. Easy readability.

9)You can load your objects on startup using lazy=false in case of Hibernate. JDBC Don't have such support.

10) Hibernate Supports automatic versioning of rows but JDBC Not.

11) Eliminates Boilerplate code.

Hibernate mapping and configuration file along with different tags and their attributes

No. of databases we are using = That many number of configuration files

hibernate.cfg.xml

```
<hibernate-configuration>
<session-factory>

<!-- Related to the connection START -->
<property name="connection.driver_class">Driver Class Name </property>
<property name="connection.url">URL </property>
<property name="connection.user">user </property>
<property name="connection.password">password</property>
<!-- Related to the connection END -->

<!-- Related to hibernate properties START -->
<property name="show_sql">true/false</property>
<property name="dialect">Database dialect class</property>
<property name="hbm2ddl.auto">create/update or whatever</property>
<!-- Related to hibernate properties END-->

<!-- Related to mapping START-->
<mapping resource="hbm file 1 name .xml" / >
<mapping resource="hbm file 2 name .xml" / >
<!-- Related to the mapping END -->
</session-factory>
</hibernate-configuration>
```

Mapping.xml

```
<hibernate-mapping>

<class name="POJO class name" table="table name in database">
<id name="variable name" column="column name in database"
type="java/hibernate type" />
<property name="variable1 name" column="column name in database"
type="java/hibernate type" />
<property name="variable2 name" column="column name in database"
type="java/hibernate type" />
</class>

</hibernate-mapping>
```

Explain session factory, is it thread safe... explain?

- ➔ Session factory objects are to be implemented using the **singleton** design pattern. Instances of SessionFactory are *thread-safe* and typically shared throughout an application. As these objects are heavy weight because they contains the connection information, hibernate configuration information and mapping files,location path. So creating number of

instances will make our application *heavy weight*. But the session objects are not thread safe. So in short it is - SessionFactory objects are one per application and Session objects are one per client.

➔ Hence it would be one SessionFactory per DataSource. Your application may have more than one DataSource so you may have more than one SessionFactory in that instance. But you would not want to create a SessionFactory more than once in an application.

➔ Session Factory is Hibernates concept of a single datastore and is threadsafe so that many threads can access it concurrently and request for sessions and immutable cache of compiled mappings for a single database. A SessionFactory is usually only built once at startup. SessionFactory should be wrapped in some kind of singleton so that it can be easily accessed in an application code.

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

Types of generators... (Primary key generation mechanism)

Generator classes are used to generate the `**identifier or primary key value**` for a persistent object while saving an object in database.

- Hibernate provides different primary key generator algorithms.
- All hibernate generator classes implements **hibernate.id.IdentifierGenerator** interface, and overrides the **generate(SessionImplementor,Object)** method to generate the `**identifier or primary key value**`.
- If we want our own user defined generator, then we should implement **IdentiferGenerator** interface and override the **generate()**
- **<generator />** tag (which is sub element of **<id />** tag) is used to configure generator class in mapping file.

Hibernate built-in generator classes

1. assigned
2. increment
3. sequence
4. hilo
5. native
6. identity
7. seqhilo
8. uuid
9. guid
10. select
11. foreign

1) Assigned

- Assigned generator class is the default generator if there is no <generator> tag and supports in all the databases.
- Developer should assign the identifier value to entity object before saving into the database.

Assigned generator configuration in hibernate mapping file.

```
<id name="empId" column="EMPNO">
   <generator class="assigned"/>
</id>
        OR
<id name="empId" column="EMPNO">
   <generator class="org.hibernate.id.Assigned"/>
</id>
        OR
<id name="empId" column="EMPNO"/>
```

2) Increment

- Increment generator supports in all databases and generates **identifier** value for new records by using below formula.

Max of Id value in Database + 1

- For first record it assigns 1 to the **identifier.** For second record it assigns based on above formula. i.e.(**Max of Id value in Database + 1) =(1+1) = 2.**

Increment generator configuration in hibernate mapping file.

```
<id name="empId" column="EMPNO">
   <generator class="increment"/>
</id>
        OR
<id name="empId" column="EMPNO">
   <generator class="org.hibernate.id.IncrementGenerator"/>
</id>
```

3) Sequence

- Sequence generator does not support MySql database and it is database dependent. i.e. Before using this generator, We should know whether this generator supports in the database or not.
- If there is no sequence in database, it uses default sequence. For ex

for oracle database it uses **HIBERNATE_SEQUENCE**

Sequence generator configuration in hibernate mapping file.

```
<id name="empId" column="EMPNO">
   <generator class="sequence">
      <param name="sequence">EMPNO_SEQ</param>
   </generator>
</id>
        OR
<id name="empId" column="EMPNO">
   <generator class="org.hibernate.id.SequenceGenerator">
      <param name="sequence">EMPNO_SEQ</param>
   </generator>
</id>
        OR
<id name="empId" column="EMPNO">
   <generator class="sequence"/>
</id>
```

4)hilo

- Database independent.
- Uses hi/lo algorithms to generate identifiers.
- It generates the identifiers based on table and column values in <generator/> tag.
- Default table is **'hibernate_unique_key'** and column is `**next_hi**`.

hilo generator configuration in hibernate mapping file

```
<id name="empId" column="EMPNO">
   <generator class="hilo">
      <param name="table">HIGH_VAL_TABLE</param>
      <param name="column">HIGH_VAL_COLUMN</param>
      <param name="max_lo">60</param>
   </generator>
</id>

        OR
<id name="empId" column="EMPNO">
   <generator class="org.hibernate.id.TableHiloGenerator">
      <param name="table">HIGH_VAL_TABLE</param>
      <param name="column">HIGH_VAL_COLUMN</param>
      <param name="max_lo">60</param>
   </generator>
```

</id>

5) native

- It uses internally **identity or sequence or hilo** generator classes.
- **native** picks up **identity or sequence or hilo** generator class depending upon the capabilities of the underlying database.

native generator configuration in hibernate mapping file

```
<id name="empId" column="EMPNO">
   <generator class="native">
      <param name="sequnce">EMPNO_SEQ</param>
      <param name="table">HIGH_VAL_TABLE</param>
      <param name="column">HIGH_VAL_COLUMN</param>
      <param name="max_lo">60</param>
   </generator>
</id>
```

6) identity

- Identity columns are support by DB2, MYSQL, SQL SERVER and SYBASE databases.

identity generator configuration in hibernate mapping file

```
<id name="empId " column="EMPNO">
      <generator class="identity"/>
</id>
```

7) seqhilo

- It is like hilo generator class, but hilo generator stores its high value in table where as seqhilo generator class stores its high value in sequence.

seqhilo configuration in hibernate mapping file

```
<id name="empId" column="EMPNO">
   <generator class="seqhilo">
      <param name="sequence">EMPNO_SEQ</param>
      <param name="max_lo">60</param>
   </generator>
</id>
```

```
        OR
<id name="empId" column="EMPNO">
   <generator class="seqhilo">
      <param
name="org.hibernate.id.SequenceHiLoGenerator">EMPNO_SEQ</para
m>
      <param name="max_lo">60</param>
   </generator>
</id>
```

8)uuid

- It uses 128-bit uuid algorithm to generate identifiers of type string.
- Generated identifier is unique within a network.
- Generates identifier using IP address.
- Encodes identifier as a string (hexadecimal digits) of length 32.
- It is used to generate passwords.

uuid configuration in hibernate mapping file

```
<id name="empId" column="EMPNO">
      <generator class="uuid">
      </generator>
</id>
        OR
<id name="empId" column="EMPNO">
      <generator class="org.hibernate.id.UUIDHexGenerator">
      </generator>
</id>
```

9) guid

- Uses a database generated guid string on MS-SQL and MY-SQL

10) select

- select retrieves a primary key assigned by a database trigger by selecting the row by some unique key and retrieving the primary key value.

11) Foreign

- foreign uses the identifier of another associated object. Usually uses in conjunction with a <one-to-one> primary key association.

getCurrentSession() vs. openSession()

Parameter	openSession	getCurrentSession
Session object	It always create new Session object	It creates a new Session if not exists , else use same session which is in current hibernate context
Flush and close	You need to explicitly flush and close session objects	You do not need to flush and close session objects, it will be automatically taken care by Hibernate internally
Performance	In single threaded environment , It is slower than getCurrentSession	In single threaded environment , It is faster than getOpenSession
Configuration	You do not need to configure any property to call this method	You need to configure additional property "hibernate.current_session_context_class" to call getCurrentSession method, otherwise it will throw exceptions.

Different between session.get() and session.load()

Actually, both functions are use to retrieve an object with different mechanism, of course.

1. session.load()

- It will always return a "**proxy**" (Hibernate term) without hitting the database. In Hibernate, proxy is an object with the given identifier value, its properties are not initialized yet, it just look like a temporary fake object.
- If no row found , it will throws an **ObjectNotFoundException**.

2. session.get()

- It always **hit the database** and return the real object, an object that represent the database row, not proxy.
- If no row found , it return **null**.

It's about performance

Hibernate create anything for some reasons, when you do the association, it's normal to obtain retrieve an object (persistent instance) from database and assign it as a reference to another object, just to maintain the relationship. Let's go through some examples to understand in what situation you should use **session.load()**.

1. session.get()-

For example, in a Stock application , Stock and StockTransactions should have a "one-to-many" relationship, when you want to save a stock transaction, it's common to declared something like below

```
Stock stock = (Stock)session.get(Stock.class, new Integer(2));
        StockTransaction stockTransactions = new StockTransaction();
        //set stockTransactions detail
        stockTransactions.setStock(stock);
        session.save(stockTransactions);
```

Output

```
Hibernate:
    select ... from mkyong.stock stock0_
    where stock0_.STOCK_ID=?
Hibernate:
    insert into mkyong.stock_transaction (...)
    values (?, ?, ?, ?, ?, ?)
```

In session.get(), Hibernate will hit the database to retrieve the Stock object and put it as a reference to StockTransaction. However, this save process is extremely high demand, there may be thousand or million transactions per hour, do you think is this necessary to hit the database to retrieve the Stock object everything save a stock transaction record? After all you just need the Stock's Id as a reference to StockTransaction.

2. session.load()

In above scenario, **session.load()** will be your good solution, let's see the

example,

```
Stock stock = (Stock)session.load(Stock.class, new Integer(2));
        StockTransaction stockTransactions = new StockTransaction();
        //set stockTransactions detail
        stockTransactions.setStock(stock);
        session.save(stockTransactions);
```

Output

```
Hibernate:
    insert into mkyong.stock_transaction (...)
    values (?, ?, ?, ?, ?, ?)
```

In session.load(), Hibernate will not hit the database (no select statement in output) to retrieve the Stock object, it will return a Stock proxy object – a fake object with given identify value. In this scenario, a proxy object is enough for to save a stock transaction record.

Exception

In exception case, see the examples

session.load()

```
Stock stock = (Stock)session.load(Stock.class, new Integer(100)); //proxy

 //initialize proxy, no row for id 100, throw ObjectNotFoundException
System.out.println(stock.getStockCode());
```

It will always return a proxy object with the given identity value, even the identity value is not exists in database. However, when you try to initialize a proxy by retrieve it's properties from database, it will hit the database with select statement. If no row is found, a **ObjectNotFoundException** will throw.

```
org.hibernate.ObjectNotFoundException: No row with the given identifier exists:
[com.mkyong.common.Stock#100]
```

session.get()

```
//return null if not found
Stock stock = (Stock)session.get(Stock.class, new Integer(100));
System.out.println(stock.getStockCode()); //java.lang.NullPointerException
```

It will always return null , if the identity value is not found in database.

Hibernate cache mechanism...explain First level, second level and Query cache...differentiate.

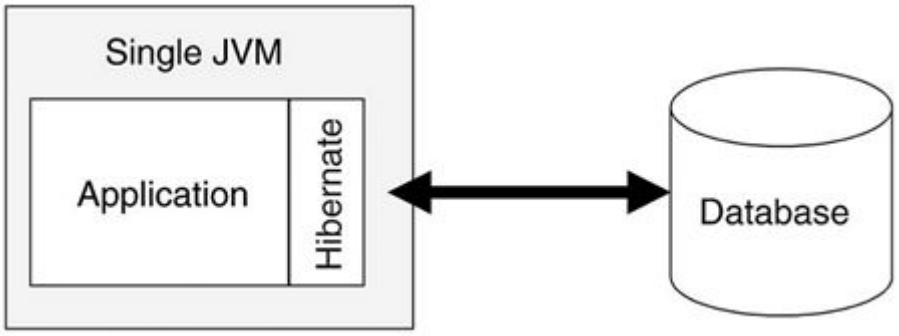
Hibernate Caching

- ➔ More the database hits, less the performance. To increase the database access performance, Hibernate comes with caching mechanism to give high performance.
- ➔ Especially, in Web applications, performance is a bottleneck due to database traffic. Caching is the most preferred technique to solve this problem. Caching is nothing but some buffer where a record is stored when first time retrieved from the database. When second time needed the same record, Hibernate does not access the database and instead reads from the cache. This type of adjustment decreases the database hits. This is what Cache Hibernate is. Accessing cache is much faster than accessing the database.

Cache Hibernate implementations

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

1. First-level cache with Session Object

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

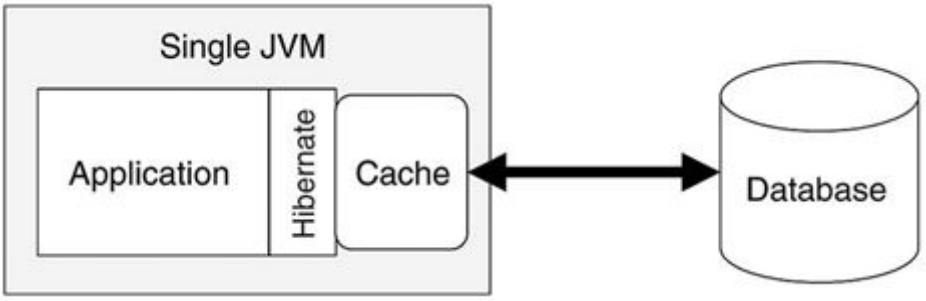
- ➔ How it works? Hibernate does not process each database hit separately and instead preserves the query in the buffer memory (associated with

Session). Infact, it writes one addBatch() statement and stores with the Session object. When the transaction is committed or session is flushed, Hibernate executes all the queries, stored earlier, with executeBatch() statement. This obviously increases the performance.

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

2. Second-level cache with SessionFactory Object

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

➔ 

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

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

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

- ➔ Each style differs in their performance, memory usage and support for different properties supported by servers like clustering etc.
- ➔ For this program of Cache Hibernate, we use EHCache. EHCache comes by default with Hibernate software.
- ➔ Here we wrote two programs with first-level and second-level caching.
- ➔ Modifications to be done in both the XML files for second-level cache usage.

a) In cfg file add following lines just before resource mapping tag as follows.

```
  <property name="cache.use_second_level_cache">true</property>
   <property
name="cache.provider_class">org.hibernate.cache.EhCacheProvider</property
>

   <mapping resource="Student.hbm.xml"/>
```

b) a) In hbm file add following line just after <class name="Studnet" ….> and just before <id name="sid" …> as follows:

```
<class name="Student" table="school">
  <cache usage="read-only"/>
  <id name="sid" column="stud_id" type="integer">
```

Let us discuss the contents of ehcache.xml file.

```
<defaultCache
  maxElementsInMemory="2" // statement informs the Hibernate to place 2
records (in Hibernate style, 2 objects) in RAM out of the many in the hard disk.
  eternal="false" // Eternal with false value indicates the records should not be
stored in hard disk permanently.
  timeToIdleSeconds="120" // 120 seconds, the records may live in RAM and
afterwards they will be moved to hard disk.
  timeToLiveSeconds="300" // The maximum storage period in the hard disk will
be 300 seconds and afterwards the records are permanently deleted from the
hard disk (not from database table).
  overflowToDisk="true" //When the records retrieved are more than 2, the
excess records may be stored in hard disk specified in the ehcahe.xml file.
/>
```

Query Cache

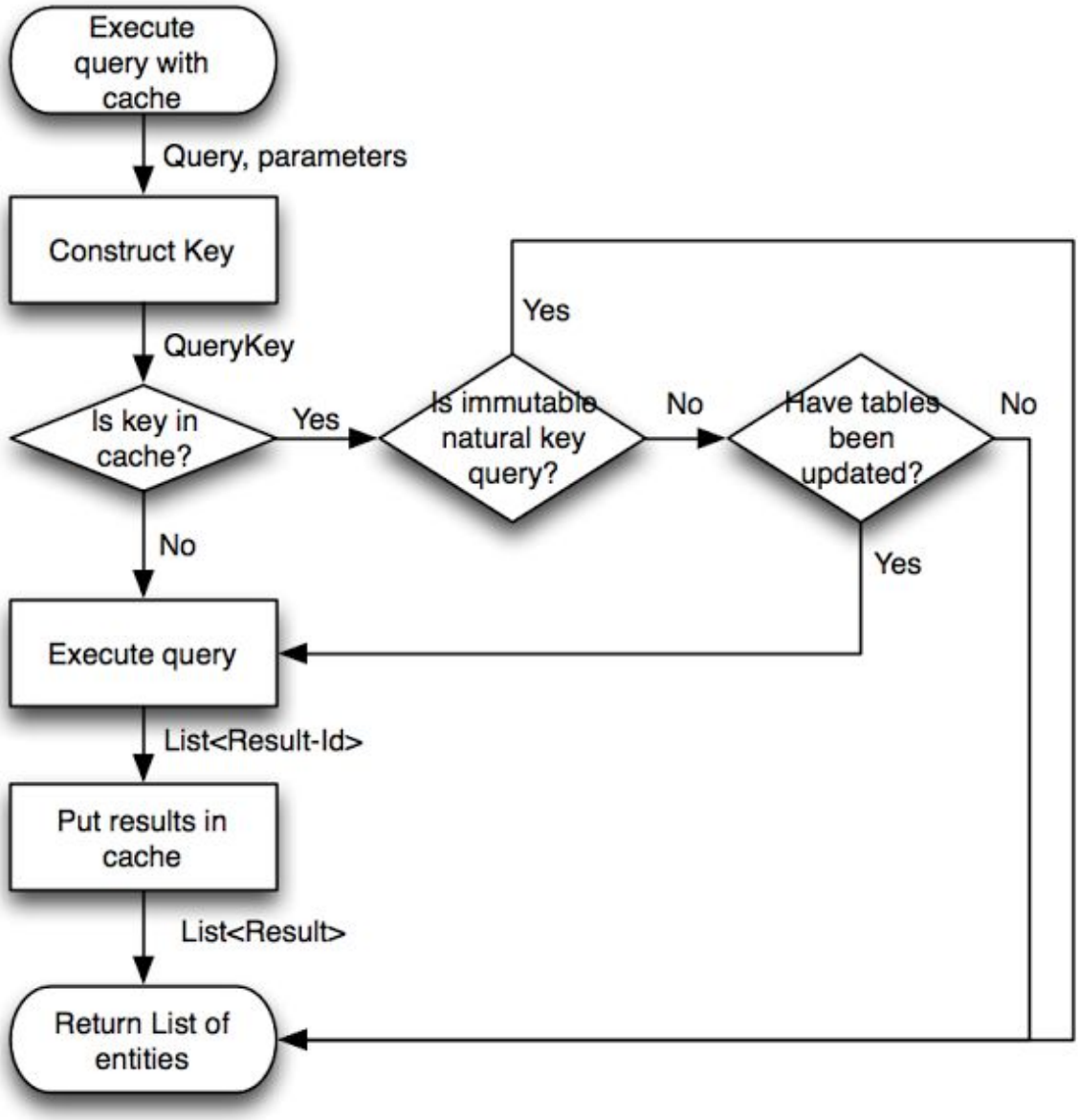
- ➔ **Query Cache** is used to cache the results of a query. When the query cache is turned on, the results of the query are stored against the combination query and parameters. Every time the query is fired the

cache manager checks for the combination of parameters and query. If the results are found in the cache, they are returned, otherwise a database transaction is initiated. As you can see, it is not a good idea to cache a query if it has a number of parameters, because then a single parameter can take a number of values. For each of these combinations the results are stored in the memory. This can lead to extensive memory usage.

- How the Query Cache Works
- The query cache is actually much like the association caching described above; it's really little more than a list of identifiers for a particular type (although again, it is much more complex internally). Let's say we performed a query like this:
- Query query = session.createQuery("from Person as p where p.parent.id=? and p.firstName=?");
  query.setInt(0, Integer.valueOf(1));
  query.setString(1, "Joey");
  query.setCacheable(true);
  List l = query.list();

- The query cache works something like this:
- *----------*
Query Cache
["from Person as p where p.parent.id=? and p.firstName=?", [1 , "Joey"]] -> [2]]

- The combination of the query and the values provided as parameters to that query is used as a key, and the value is the list of identifiers for that query. Note that this becomes more complex from an internal perspective as you begin to consider that a query can have an effect of altering associations to objects returned that query; not to mention the fact that a query may not return whole objects, but may in fact only return scalar values (when you have supplied a select clause for instance). That being said, this is a sound and reliable way to think of the query cache conceptually.
- If the second-level cache is enabled (which it should be for objects returned via a query cache), then the object of type Person with an id of 2 will be pulled from the cache (if available in the cache), it will then be hydradated, as well as any associations.



Types of Hibernate instance states - transient, persistent and detached explain

1. Transient State:

A New instance of a persistent class which is not associated with a ***Session***, has no representation in the ***database*** and no identifier value is considered ***transient*** by Hibernate:

```
1. UserDetail user = new UserDetail();
2.
3. user.setUserName("Dinesh Rajput");
4.
5. // user is in a transient state
```

2. Persistent State:

A persistent instance has a representation in the ***database*** , an identifier

value and is associated with a ***Session***. You can make a transient instance persistent by associating it with a ***Session***:

```
1. Long id = (Long) session.save(user);
2.
3. // user is now in a persistent state
```

3. Detached State:

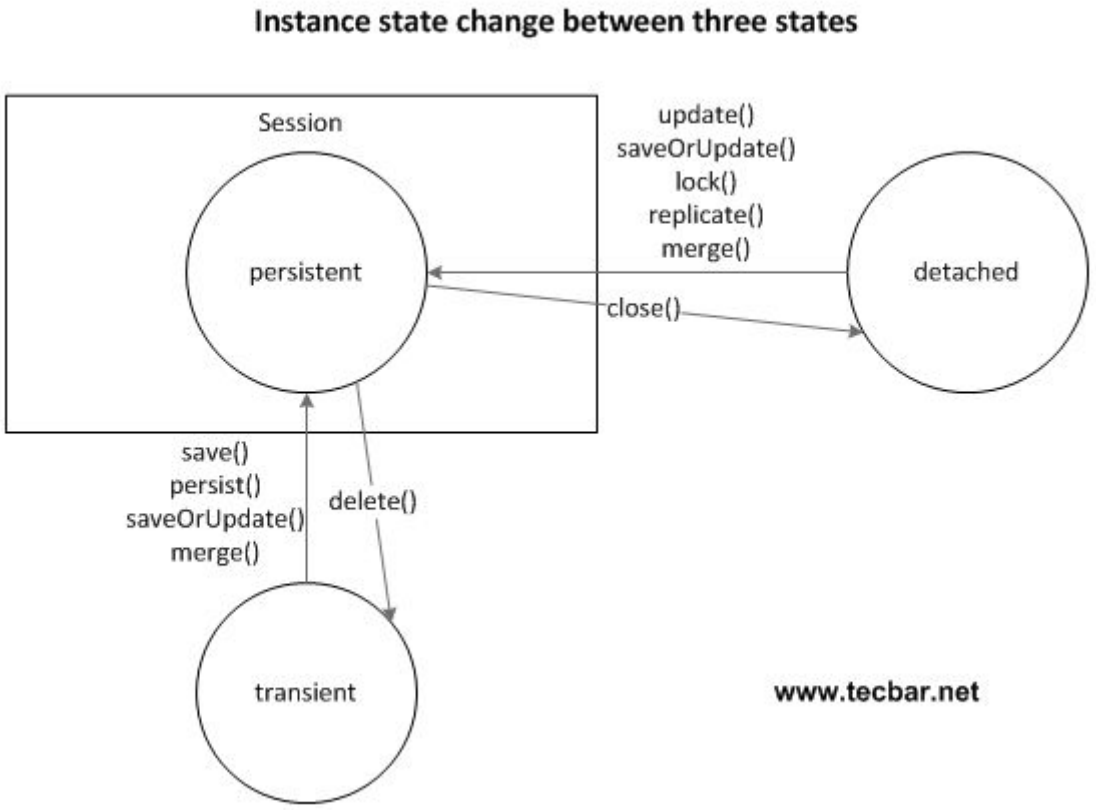
Now, if we close the ***Hibernate Session***, the ***persistent*** instance will become a ***detached*** instance: it isn't attached to a ***Session*** anymore (but can still be modified and reattached to a new Session later though).

```
1. session.close();
2.
3. //user in detached state
```

Difference between Transient and Detached States:

- ➔ Transient objects do not have association with the databases and session objects. They are simple objects and not persisted to the database. Once the last reference is lost, that means the object itself is lost. And of course , garbage collected. The commits and rollbacks will have no effects on these objects. They can become into persistent objects through the save method calls of Session object.

- ➔ The detached object have corresponding entries in the database. These are persistent and not connected to the Session object. These objects have the synchronized data with the database when the session was closed. Since then, the change may be done in the database which makes this object stale. The detached object can be reattached after certain time to another object in order to become persistent again.



Inheritance models in Hibernate

- Table per class hierarchy

EDIT	ID	TYPE	NAME	SALARY	BONUS	PAY_PER_HOUR	CONTRACT_DURATION
	1	emp	sonoo	-	-	-	-
	2	reg_emp	Vivek Kumar	50000	5	-	-
	3	con_emp	Arjun Kumar	-	-	1000	15 hours
							row(s) 1 - 3 of 3

Annotations:-

@Inheritance(strategy=InheritanceType.SINGLE_TABLE)

@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)

@DiscriminatorValue(value="employee")

- ➔ In Single table per subclass, the union of all the properties from the inheritance hierarchy is mapped to one table. As all the data goes in one table, a discriminator is used to differentiate between different type of data.

➔ **Advantages of Single Table per class hierarchy**

- Simplest to implement.
- Only one table to deal with.
- Performance wise better than all strategies because no joins or sub-selects need to be performed.

➔ **Disadvantages:**

- Most of the column of table are nullable so the NOT NULL constraint cannot be applied.
- Tables are not normalized.

- Table per concrete class

Employee Class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
				1 - 2

Regular Employee:-

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
				1 - 4

Contract Employee:-

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-
				1 - 4

Annotations: @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)

➔ In this case every entity class has its own table i.e. table per class. The

data for Vehicle is duplicated in both the tables.

➔ This strategy is not popular and also have been made optional in Java Persistence API.

➔ **Advantage:**

- Possible to define NOT NULL constraints on the table.

➔ **Disadvantage:**

- Tables are not normalized.
- To support polymorphism either container has to do multiple trips to database or use SQL UNION kind of feature.

In this case there no need for the discriminator column because all entity has own table.

- Table per subclass

Table structure for Employee class

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER(10,0)	No	-	1
NAME	VARCHAR2(255)	Yes	-	-
				1 - 2

Table structure for Regular_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
EID	NUMBER(10,0)	No	-	1
SALARY	FLOAT	Yes	-	-
BONUS	NUMBER(10,0)	Yes	-	-
				1 - 3

Table structure for Contract_Employee class

Column Name	Data Type	Nullable	Default	Primary Key
EID	NUMBER(10,0)	No	-	1
PAY_PER_HOUR	FLOAT	Yes	-	-
CONTRACT_DURATION	VARCHAR2(255)	Yes	-	-
				1 - 3

Annotations: @Inheritance(strategy=InheritanceType.JOINED)

It's highly normalized but performance is not good.

➔ **Advantage:**

- Tables are normalized.
- Able to define NOT NULL constraint.

➔ **Disadvantage:**

- Does not perform as well as SINGLE_TABLE strategy

Explain session method - save, persist, update, saveorupdate merge etc

Hibernate Session save

➔ As the method name suggests, hibernate save() can be used to save entity to database. We can invoke this method outside a transaction, that's why I don't like this method to save data. If we use this without transaction and we have cascading between entities, then only the primary entity gets saved unless we flush the session.

Few important points that we can confirm from output are:

- We should avoid save outside transaction boundary, otherwise mapped entities will not be saved causing data inconsistency. It's very normal to forget flushing the session because it doesn't throw any exception or warnings.
- Hibernate save method returns the generated id immediately, this is possible because primary object is saved as soon as save method is invoked.
- If there are other objects mapped from the primary object, they gets saved at the time of committing transaction or when we flush the session.
- For objects that are in persistent state, save updates the data through update query. Notice that it happens when transaction is committed. If there are no changes in the object, there wont be any query fired. If you will run above program multiple times, you will notice that update queries are not fired next time because there is no change in the column values.
- Hibernate save load entity object to persistent context, if you will update the object properties after the save call but before the transaction is committed, it will be saved into database.

```
//save example - without transaction
                Session session = sessionFactory.openSession();
                Employee emp = getTestEmployee();
                long id = (Long) session.save(emp);
                System.out.println("1. Employee save called without transaction,
id="+id);
                session.flush(); //address will not get saved without this
                System.out.println("*****");

                //save example - with transaction
```

```
Transaction tx1 = session.beginTransaction();
Session session1 = sessionFactory.openSession();
Employee emp1 = getTestEmployee();
long id1 = (Long) session1.save(emp1);
System.out.println("2. Employee save called with transaction,
id="+id1);
System.out.println("3. Before committing save transaction");
tx1.commit();
System.out.println("4. After committing save transaction");
System.out.println("*****");
```

Hibernate persist

- ➔ It is similar to save (with transaction) and it adds the entity object to the **persistent context**, so any further changes are tracked. If the object properties are changed before the transaction is committed or session is flushed, it will also be saved into database.
- ➔ Second difference is that we can use persist() method only within the boundary of a transaction, so it's safe and takes care of any cascaded objects.
- ➔ Finally, persist doesn't return anything so we need to use the persisted object to get the generated identifier value.

```
Session session2 = sessionFactory.openSession();
Transaction tx2 = session2.beginTransaction();
Employee emp2 = HibernateSaveExample.getTestEmployee();
session2.persist(emp2);
System.out.println("Persist called");
emp2.setName("Kumar"); // will be updated in database too
System.out.println("Employee Name updated");
System.out.println("8. Employee persist called with transaction,
id="+emp2.getId()+", address id="+emp2.getAddress().getId());
tx2.commit();
System.out.println("*****");

// Close resources
sessionFactory.close();
```

Hibernate saveOrUpdate

- ➔ Hibernate saveOrUpdate results into insert or update queries based on the provided data. If the data is present in the database, update query is executed.
- ➔ We can use saveOrUpdate() without transaction also, but again you will face the issues with mapped objects not getting saved if session is not flushed.

- ➔ Hibernate saveOrUpdate adds the entity object to persistent context and track any further changes. Any further changes are saved at the time of committing transaction, like persist.

```
//saveOrUpdate example - without transaction
		Session session5 = sessionFactory.openSession();
		Employee emp5 = HibernateSaveExample.getTestEmployee();
		session5.saveOrUpdate(emp5);
		System.out.println("*****");

		//saveOrUpdate example - with transaction
		Session session3 = sessionFactory.openSession();
		Transaction tx3 = session3.beginTransaction();
		Employee emp3 = HibernateSaveExample.getTestEmployee();
		session3.saveOrUpdate(emp3);
		emp3.setName("Kumar"); //will be saved into DB
		System.out.println("9. Before committing saveOrUpdate
transaction. Id="+emp3.getId());
		tx3.commit();
		System.out.println("10. After committing saveOrUpdate
transaction");
		System.out.println("*****");


		Transaction tx4 = session3.beginTransaction();
		emp3.setName("Updated Test Name"); //Name changed
		emp3.getAddress().setCity("Updated City");
		session3.saveOrUpdate(emp3);
		emp3.setName("Kumar"); //again changed to previous value, so no
Employee update
		System.out.println("11. Before committing saveOrUpdate
transaction. Id="+emp3.getId());
		tx4.commit();
		System.out.println("12. After committing saveOrUpdate
transaction");
		System.out.println("*****");

		// Close resources
		sessionFactory.close();
```

- ➔ Notice that without transaction, only Employee gets saved and address information is lost.
- ➔ With transaction employee object is tracked for any changes, thats why in last call there is no update in Employee table even though the value was changed in between, final value remains same.

Hibernate update

- ➔ Hibernate update should be used where we know that we are only updating the entity information. This operation adds the entity object to persistent context and further changes are tracked and saved when transaction is committed.

- ➔ Session session = sessionFactory.openSession();
  Transaction tx = session.beginTransaction();
  Employee emp = (Employee) session.load(Employee.class, new Long(101));
  System.out.println("Employee object loaded. " + emp);
  tx.commit();

  // update example
  emp.setName("Updated name");
  emp.getAddress().setCity("Bangalore");
  Transaction tx7 = session.beginTransaction();
  session.update(emp);
  emp.setName("Final updated name");
  System.out.println("13. Before committing update transaction");
  tx7.commit();
  System.out.println("14. After committing update transaction");

  // Close resources
  sessionFactory.close();
- ➔ Notice that there are no updates fired after first execution because there are no update in values. Also notice the employee name is "Final updated name" that we set after invoking update() method. This confirms that hibernate was tracking the object for any changes and at the time of committing transaction, this value got saved.

Hibernate Merge

- ➔ **Hibernate merge** can be used to update existing values, however this method create a copy from the passed entity object and return it. The returned object is part of persistent context and tracked for any changes, passed object is not tracked. This is the major difference with merge() from all other methods.

```
SessionFactory sessionFactory = HibernateUtil.getSessionFactory();
        Session session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();
        Employee emp = (Employee) session.load(Employee.class, new Long(101));
```

```
System.out.println("Employee object loaded. " + emp);
tx.commit();

//merge example - data already present in tables
emp.setSalary(25000);
Transaction tx8 = session.beginTransaction();
Employee emp4 = (Employee) session.merge(emp);
System.out.println(emp4 == emp); // returns false
emp.setName("Test");
emp4.setName("Kumar");
System.out.println("15. Before committing merge transaction");
tx8.commit();
System.out.println("16. After committing merge transaction");

// Close resources
sessionFactory.close();
```

➔ Notice that the entity object returned by merge() is different from the passed entity. Also notice that in further execution, name is "Kumar", this is because the returned object is tracked for any changes.

Explain following hibernate properties

1. show_sql

- Enable the logging of all the generated SQL statements to the console
- generates sql statements to the console.
- for ex;ample,

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

2. hbm2ddl.auto (Create-drop, update, validate etc)

- hbm2ddl configuration means hibernate mapping to create schema ddl.
- **create -**

schema ddl created every time when sessionfactory object is created.

- **create-drop -**

the database schema will be dropped when the sessionfactory is explicitly closed.

- **update -**

update existing schema.

- **validate -**

validate that the schema matches, make no changes to the schema of the database, you probably want this for production.

3. Dialect

- database specific.
- if we connecting any hibernate application with database we should specify the dialect in configuration file.
- for example- mysql-org.hibernate.dialect.MySQLDialect

oracle- org.hibernate.dialect.OracleDialect

4. connection pool

Connection pool is good for performance, as it prevents Java application create a connection each time when interact with database and minimizes the cost of opening and closing connections.

1. hibernate.c3p0.min_size – Minimum number of JDBC connections in the pool. Hibernate default: 1
2. hibernate.c3p0.max_size – Maximum number of JDBC connections in the pool. Hibernate default: 100
3. hibernate.c3p0.timeout – When an idle connection is removed from the pool
4. (in second). Hibernate default: 0, never expire.
5. hibernate.c3p0.max_statements – Number of prepared statements will be cached. Increase performance. Hibernate default: 0 , caching is disable.
6. hibernate.c3p0.idle_test_period – idle time in seconds before a connection is automatically validated. Hibernate default: 0

Differences with coding example

- Save VS SaveorUPdate vs. Persist

Parameter	Save	SaveOrUpadate	Persist
return type.	Serializable object.		void
transaction boundaries.	save() method does not guarantee the same, it returns an identifier, and if an INSERT has to be executed to get the identifier (e.g. "identity" generator), this INSERT happens immediately, no matter if you are inside or outside of a transaction.	saveOrUpdate can either INSERT or UPDATE based upon existence of record.If persistence object already exists in database then UPDATE SQL will execute and if there is no corresponding object in database than INSERT will run.	persist() method guarantees that it will not execute an INSERT statement if it is called outside of transaction boundaries.

Application	On the other hand save method is not good in a long-running conversation with an extended Session context.	saveOrUpdate is more flexible in terms of use but it involves an extra processing to find out whether record already exists in table or not.	Because of its above behavior of persist method outside transaction boundary, its useful in long-running conversations with an extended Session context.

- Update vs. Merge

update	merge
When the session does not contain the persistent instance with the same identifier, and if it is sure use update for the data persistence in the hibernate.	Irrespective of the state of a session, if there is a need to save the modifications at any given time, use merge().
Update should be used to save the data when the session does not contain an already persistent instance with the same identifier.	Merge should be used to save the modifications at any time without knowing about the state of a session.
if object does not exist in db already then the update() method will throw the exception	if object does not exist in db already then the merge() will create the new entry in db for that object.

Collection types in Hibernate -Set, List, Array, Map, and Bag

- ➔ A **Bag** is a java collection that stores elements without caring about the sequencing but allow duplicate elements in the list. A bag is a random grouping of the objects in the list.
- ➔ A Collection is mapped with a <bag> element in the mapping table and initialized with java.util.ArrayList.

```
<bag name="certificates" cascade="all">
        <key column="employee_id"/>
        <one-to-many class="Certificate"/>
</bag>
```

Array:

```
<array name=”arrayName” table=”tableName” cascade=”all”>
<key column=”anyColumnName” />
<list-index column=”anyColumnName” base=”1”>
</list-index>
<many-to-many column=”anyColumnName” unique=”true” class=”className”
/>
</array>
```

- ➔ array name – name matches with array name in java.
- ➔ Key column – any unique column name for the given table.
- ➔ list-index column – any column name which is used to map the tables.
- ➔ Base=”1” – Its optional. It’s default value is zero. List starts from one so, here base=”1”.
- ➔ Many-to-many column – Column name which is used to map the table with other table.

- ➔ A **Set** is a java collection that does not contain any duplicate element. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. So objects to be added to a set must implement both the equals() and hashCode() methods so that Java can determine whether any two elements/objects are identical.
- ➔ A Set is mapped with a <set> element in the mapping table and initialized with java.util.HashSet. You can use Set collection in your class when there is no duplicate element required in the collection.

```
<list name="certificates" cascade="all">
            <key column="employee_id"/>
            <list-index column="idx"/>
            <one-to-many class="Certificate"/>
 </list>
```

- ➔ A **List** is a java collection that stores elements in sequence and allow duplicate elements. The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index, and search for elements in the list. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all.

- ➔ A List is mapped with a <list> element in the mapping table and initialized with java.util.ArrayList.

```
<list name="certificates" cascade="all">
        <key column="employee_id"/>
```

```
        <list-index column="idx"/>
        <one-to-many class="Certificate"/>
    </list>
```

- ➔ A **Map** is a java collection that stores elements in key-value pairs and does not allow duplicate elements in the list. The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.
- ➔ A Map is mapped with a <map> element in the mapping table and an unordered map can be initialized with java.util.HashMap.

```
<map name="certificates" cascade="all">
        <key column="employee_id"/>
        <index column="certificate_type" type="string"/>
        <one-to-many class="Certificate"/>
</map>
```

Difference between sorted and ordered collection in hibernate?
Sorted Collection :

- The sorted collection is a collection that is sorted using the Java collections framework. The sorting is done in the memory of JVM that is running hibernate, soon after reading the data from the database using Java Comparator
- The less the collection the more the efficient of sorting

Ordered Collection :

- The order collections will also sorts a collection by using the order by clause for the results fetched.
- The more the collection, the more efficient of sorting.

the difference between sorted and ordered collection in hibernate

- A sorted collection is sorted 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.
- The efficiency depends on the size of the collection.
- Ordered collection is sorted by specifying the order-by clause for sorting this collection when retrieval.
- This is an efficient way to sort larger collections.

Explain Named and native hibernate queries, differentiate
What is the Difference between Native Query and Named Native Query :
here I will give a short notice about the difference between them

Native Query :

- ➔ the developer is writing a native SQL query. and has and i'ts dynamic query , it's compiled and execute in run time and will be validated each time the query is executed.

Named Native Query :

- ➔ the scope is persistence ontext and can be used in the application by Specifying the Identity , and it is ready defined and Compiled once , and it's of course static statement.
- ➔ they are validated only once, at the server startup

Explain criteria API along with examples

- ➔ Criteria is a simplified API for retrieving entities by composing Criterion objects. This is a very convenient approach for functionality like "search" screens where there is a variable number of conditions to be placed upon the result set.

- ➔ The Session is a factory for Criteria. Criterion instances are usually obtained via the factory methods on Restrictions.

Example:

```
List cats = session.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Iz%") )
    .add( Restrictions.gt( "weight", new Float(minWeight) ) )
    .addOrder( Order.asc("age") )
    .list();
```

What is hibernate template

- ➔ HibernateTemplate is a helper class that is used to simplify the data access code. This class supports automatically converts HibernateExceptions which is a checked exception into DataAccessExceptions which is an unchecked exception. HibernateTemplate is typically used to implement data access or business logic services. The central method is execute(), that supports the Hibernate code that implements HibernateCallback interface.

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.

- **The benefits of HibernateTemplate are:**
1. HibernateTemplate, which is a Spring Template class, can simplify the interactions with Hibernate Sessions.
2. Various common functions are simplified into single method invocations.
3. The sessions of hibernate are closed automatically
4. The exceptions will be caught automatically, and converts them into runtime exceptions.

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.

- ➔ Required to set the **hibernate.dialect** property to the correct **org.hibernate.dialect.Dialect** subclass for your database. If you specify a dialect, Hibernate will use sensible defaults for some of the other properties listed above. This means that you will not have to specify them manually.

RDBMS	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Sybase	org.hibernate.dialect.SybaseDialect

Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
Firebird	org.hibernate.dialect.FirebirdDialect

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

- ➔ You can add properties direct to hibernate.cfg.xml like this :
- ➔ `<property name="show_sql">true</property>`

For Eclipse users:

You can turn on hibernate.show_sql=true in MyEclipse config editor - open your hibernate.cf.xml, find tab 'Properties' click add, choose
show_sql and set value to true

Define cascade and inverse option in one-many mapping

1. inverse

This is used to decide which side is the relationship owner to manage the relationship (insert or update of the foreign key column).

2. cascade

In cascade, after one operation (save, update and delete) is done, it decide whether it need to call other operations (save, update and delete) on another entities which has relationship with each other.

cascade - enable operations to cascade to child entities.
cascade="all|none|save-update|delete|all-delete-orphan"

inverse - mark this collection as the "inverse" end of a bidirectional association.
inverse="true|false"

Essentially "inverse" indicates which end of a relationship should be ignored, so when persisting a parent who has a collection of children, should you ask the parent for its list of children, or ask the children who the parents are?

Explain mapping file id field

Mapped classes *must* declare the primary key column of the database table. Most classes will also have a JavaBeans-style property holding the unique identifier of an instance. The <id> element defines the mapping from that property to the primary key column.

```
<id
        name="propertyName"                                    1
        type="typename"                                2
        column="column_name"                                   3
        unsaved-value="null|any|none|undefined|id_value"                4
        access="field|property|ClassName">                             5
        node="element-name|@attribute-name|element/@attribute|."

        <generator class="generatorClass"/>
</id>
```

name (optional): the name of the identifier property.
type (optional): a name that indicates the Hibernate type.
column (optional - defaults to the property name): the name of the primary key column.
unsaved-value (optional - defaults to a "sensible" value): an identifier property value that indicates an instance is newly instantiated (unsaved), distinguishing it from detached instances that were saved or loaded in a previous session.
access (optional - defaults to property): the strategy Hibernate should use for accessing the property value.
If the name attribute is missing, it is assumed that the class has no identifier property.
The unsaved-value attribute is almost never needed in Hibernate3.
There is an alternative <composite-id> declaration that allows access to legacy data with composite keys. Its use is strongly discouraged for anything else.

What is dirty checking?

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.
The automatic dirty checking feature of hibernate, calls update statement automatically on the objects that are modified in a transaction.
Let's understand it by the example given below:

```
1. ...
2. SessionFactory factory = cfg.buildSessionFactory();
3. Session session1 = factory.openSession();
4. Transaction tx=session2.beginTransaction();
5.
6. Employee e1 = (Employee) session1.get(Employee.class,
   Integer.valueOf(101));
7.
8. e1.setSalary(70000);
9.
10.tx.commit();
11.session1.close();
```

Here, after getting employee instance e1 and we are changing the state of e1. After changing the state, we are committing the transaction. In such case, state will be updated automatically. This is known as dirty checking in hibernate.

What are Call-back 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.
- ➔ 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 /as annotation. This forces hibernate to bypass the setter method and access the instance variable directly while initializing a newly loaded object.
- ➔ Name some important annotations used for Hibernate mapping..explain..
- ➔ Hibernate supports JPA annotations and it has some other annotations in org.hibernate.annotations package. Some of the important JPA and hibernate annotations used are :-

1. javax.persistence.Entity: Used with model classes to specify that they are entity beans.
2. javax.persistence.Table: Used with entity beans to define the corresponding table name in database.
3. javax.persistence.Access: Used to define the access type, either field or property. Default value is field and if you want hibernate to use getter/setter methods then you need to set it to property.
4. javax.persistence.Id: Used to define the primary key in the entity bean.
5. javax.persistence.EmbeddedId: Used to define composite primary key in

the entity bean.

6. javax.persistence.Column: Used to define the column name in database table.
7. javax.persistence.GeneratedValue: Used to define the strategy to be used for generation of primary key. Used in conjunction with javax.persistence.GenerationType enum.
8. javax.persistence.OneToOne: Used to define the one-to-one mapping between two entity beans. We have other similar annotations as OneToMany, ManyToOne and ManyToMany.
9. org.hibernate.annotations.Cascade: Used to define the cascading between two entity beans, used with mappings. It works in conjunction with org.hibernate.annotations.CascadeType.
10. javax.persistence.PrimaryKeyJoinColumn: Used to define the property for foreign key. Used with org.hibernate.annotations.GenericGenerator and org.hibernate.annotations.Parameter.

Write code to configure Hibernate Second Level Cache using EHCache?

<property key="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>
<property name="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</property>

By default, Ehcache will create separate cache regions for each entity that you configure for caching. You can change the defaults for these regions by adding the configuration to your ehcache.xml. To provide this configuration file, use this property in hibernate configuration:

<property name="net.sf.ehcache.configurationResourceName">/ehcache.xml</property>

What will happen if we don't have no-args constructor in Entity bean?

Hibernate uses Reflection API to create instance of Entity beans, usually when you call get() or load() methods. The method `Class.newInstance()` is used for this and it requires no-args constructor. So if you won't have no-args constructor in entity beans, hibernate will fail to instantiate it and you will get `HibernateException`.

How to implement Joins in Hibernate?

INNER JOIN

Query:-

```
String hql = "from Company as comp inner join comp.employees as emp";
```

HQL JOIN DEMO

```
package com.concretepage;
import java.util.List;
import org.hibernate.Session;
public class HQLJoinDemo {
	public static void main(String[] args) {
		Session session = HibernateUtil.getSessionFactory().openSession();
		String hql = "from Company as comp inner join comp.employees as
emp";
		List<?> list = session.createQuery(hql).list();
		for(int i=0; i<list.size(); i++) {
			Object[] row = (Object[]) list.get(i);
			Company company = (Company)row[0];
			Employee employee = (Employee)row[1];
			System.out.println("CompId:"+company.getCompId()+",
CompName:"+ company.getCompName()+
					", EmpId:"+ employee.getEmpId()+",
EmpName:"+ employee.getEmpName());
		}
		session.close();
	}
}
```

LEFT OUTER JOIN

```
String hql = "from Company as comp left outer join comp.employees as emp";
```

RIGHT OUTER JOIN

```
String hql = "from Company as comp right outer join comp.employees as emp";
```

CROSS JOIN

```
String hql = "from Company as comp, Employee as emp";
```

Why we should not make Entity Class final?

- ➔ By definition, an entity is a class that can be persisted to a database, so having a final field would make no sense in the context of something that will end up being a record in your database. The requirement of no final class and no final methods has to do with the way in which JPA implementations actually deal with persisting instances of your entities classes.
- ➔ It is common for implementations to subclass your class at runtime and/or add byte code dynamically to your classes, in order to detect when a change to an entity object has occurred. If your class is declared as final, then that would not be possible.
- ➔ Hibernate use proxy classes for lazy loading of data, only when it's needed. This is done by extending the entity bean, if the entity bean will be final then lazy loading will not be possible, hence low performance.

What is HQL and what are its benefits?

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

benefits -

- ➔ HQL is database independent, means if we write any program using HQL commands then our program will be able to execute in all the databases without doing any further changes to it
- ➔ HQL supports object oriented features like ***Inheritance***, ***polymorphism***, ***Associations*** (Relation ships)
- ➔ HQL is initially given for selecting object from database and in hibernate 3.x we can do DML operations (insert, update…) too

Can we execute native sql query in hibernate?

Yes, Obviously.

- By using Native SQL, we can perform both select, non-select operations on the data
- In face Native SQL means using the direct SQL command specific to the particular (current using) database and executing it with using hibernate

Advantages and Disadvantages of Native SQL

- We can use the database specific keywords (commands), to get the data from the database
- While migrating a JDBC program into hibernate, the task becomes very simple because JDBC uses direct SQL commands and hibernate also supports the same commands by using this Native SQL
- The main drawbacks of Native SQL is, some times it makes the hibernate application as database dependent one
- If we want to execute Native SQL Queries on the database then, we need to construct an object of SQLQuery, actually this SQLQuery is an interface extended from Query and it is given in ” org.hibernate package ”
- In order to get an object of SQLQuery, we need to use a method createSQLQuery() given by session interface.
- While executing native sql queries on the database, we use directly tables, column names directly in our command.
- **Remember**, while executing Native SQL Queries, even though we are

selecting complete objects from the database we need to typecast into object array only, not into our pojo class type, because we are giving direct table, column names in the Native SQL Query so it does'nt know our class name

- If we execute the command, always first it will puts data in **ResultSet** and from there List

How to log hibernate generated sql queries in log files?

<property name="show_sql">true</property>
or log4j.logger.org.hibernate.SQL=DEBUG
log4j.logger.org.hibernate.type=TRACE

What is Hibernate Proxy and how it helps in lazy loading?

➔ Proxies are classes generated dynamically by Hibernate to help with lazy loading. For instance, if you have a Cat class, Hibernate will generate a proxy class that extends Cat.
➔ If you get an uninitialized instance of this proxy, essentially all its fields will be null except the ID because Hibernate has not yet hit the database. Now the first time you will call a method on this proxy, it will realize that it is not initialized and it will query the database to load it's attributes. This is possible because the dynamically generated class overrides the base class's methods and adds this initialized/uninitialized check.
➔ Now assume that your Cat class is not a proxy and that it has a father association, when you load a Cat object, Hibernate will need to load all it's attributes. So if you load a Cat object, Hibernate will also need to load its father and the father's father and so on. Using proxies enable Hibernate to only load the required instances.
➔ Hibernate uses proxy object to support lazy loading. Basically when you load data from tables, hibernate doesn't load all the mapped objects.
➔ As soon as you reference a child or lookup object via getter methods, if the linked entity is not in the session cache, then the proxy code will go to the database and load the linked object. It uses javassist to effectively and dynamically generate sub-classed implementations of your entity objects.

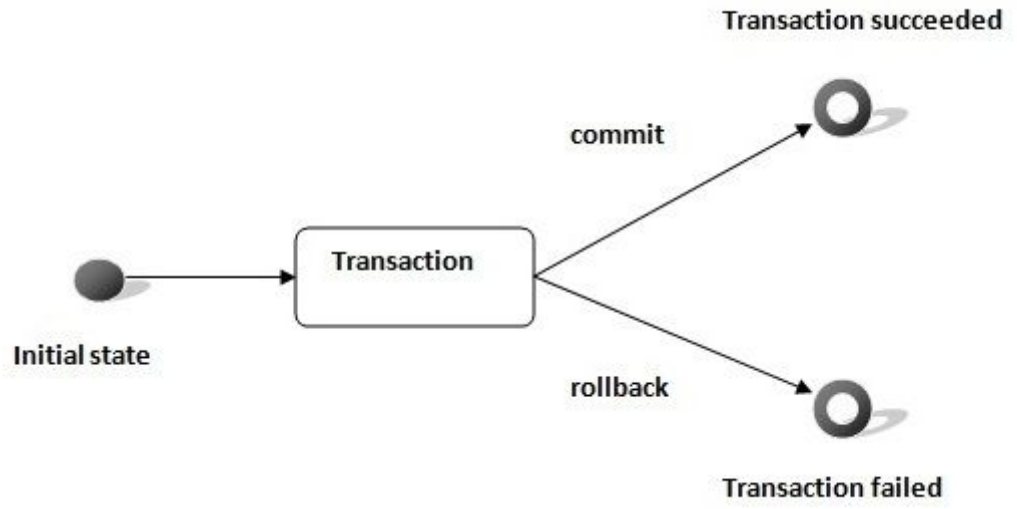
How to implement relationships in hibernate?

➔ The mapping of associations between entity classes and the relationships between tables is the soul of ORM. Following are the four ways in which the cardinality of the relationship between the objects can be expressed. An association mapping can be unidirectional as well as bidirectional.

Mapping type	Description
Many-to-One	Mapping many-to-one relationship using Hibernate
One-to-One	Mapping one-to-one relationship using Hibernate
One-to-Many	Mapping one-to-many relationship using Hibernate
Many-to-Many	Mapping many-to-many relationship using Hibernate

How transaction management works in Hibernate?

➔ A **transaction** simply represents a unit of work. In such case, if one step fails, the whole transaction fails (which is termed as atomicity). A transaction can be described by ACID properties (Atomicity, Consistency, Isolation and Durability).



Transaction Interface in Hibernate

In hibernate framework, we have **Transaction** interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA,JDBC).
A transaction is associated with Session and instantiated by calling **session.beginTransaction()**.
The methods of Transaction interface are as follows:

1. **void begin()** starts a new transaction.
2. **void commit()** ends the unit of work unless we are in FlushMode.NEVER.
3. **void rollback()** forces this transaction to rollback.
4. **void setTimeout(int seconds)** it sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
5. **boolean isAlive()** checks if the transaction is still alive.
6. **void registerSynchronization(Synchronization s)** registers a user synchronization callback for this transaction.
7. **boolean wasCommited()** checks if the transaction is commited successfully.
8. **boolean wasRolledBack()** checks if the transaction is rolledback successfully.

Example of Transaction Management in Hibernate

In hibernate, it is better to rollback the transaction if any exception occurs, so that resources can be free. Let's see the example of transaction management in hibernate.

```
Session session = null;
Transaction tx = null;
try {
session = sessionFactory.openSession();
tx = session.beginTransaction();
//some action
tx.commit();
}catch (Exception ex) {
ex.printStackTrace();
tx.rollback();
```

}

finally {session.close();}

What is cascading and what are different types of cascading?

- Cascade attribute is mandatory, when ever we apply relationship between objects, cascade attribute transfers operations done on one object onto its related child objects
- If we write cascade = "all" then changes at parent class object will be effected to child class object too, if we write cascade = "all" then all operations like insert, delete, update at parent object will be effected to child object also
- Example: if we apply insert(or update or delete) operation on parent class object, then child class objects will also be stored into the database.
- default value of cascade ="none" means no operations will be transfers to the child class.

Example:
if we apply insert(or update or delete) operation on parent class object, then child class objects will not be effected, if cascade = "none"
Cascade having the values…….

- none (default)
- save
- update
- save-update
- delete
- all
- all-delete-orphan
- In hibernate relations, if we load one parent object from the database then child objects related to that parent object will be loaded into one collection right (see one-to-many insert example).
- Now if we delete one child object from that collection, then the relationship between the parent object and that child object will be removed, but the record (object) in the database will remains at it is, so if we load the same parent object again then this deleted child will not be loaded [but it will be available on the database]
- so finally what am saying is all-delete-orphan means, breaking relation between objects not deleting the objects from the database, hope you got what am saying
- **Note**:
- what is orphan record ..?
- an orphan record means it is a record in child table but it doesn't have association with its parent in the application.
- In an application, if a child record is removed from the collection and if we want to remove that child record immediately from the database, then we

need to set the cascade =”all-delete-orphan”

- And that’s it about this cascade attribute in hibernate, hope i explained all the values..!!

@Annotations

The cascade types supported by the Java Persistence Architecture are as below:

1. **CascadeType.PERSIST** : means that save() or persist() operations cascade to related entities.
2. **CascadeType.MERGE** : means that related entities are merged when the owning entity is merged.
3. **CascadeType.REFRESH** : does the same thing for the refresh() operation.
4. **CascadeType.REMOVE** : removes all related entities association with this setting when the owning entity is deleted.
5. **CascadeType.DETACH** : detaches all related entities if a “manual detach” occurs.
6. **CascadeType.ALL** : is shorthand for all of the above cascade operations.

The cascade configuration option accepts an array of CascadeTypes.

Advantage of Spring framework with hibernate

The Spring framework provides **HibernateTemplate** class, so you don't need to follow so many steps like create Configuration, BuildSessionFactory, Session, beginning and committing transaction etc. So **it saves a lot of code**. MAPPING:

<beanid="template"class="org.springframework.orm.hibernate3.HibernateTemp

late">

<property name="sessionFactory" ref="mysessionFactory"></property>

</bean>

<bean id="d" class="com.javatpoint.EmployeeDao">

<property name="template" ref="template"></property>

</bean>

Which design patterns are used in Hibernate framework?

- ➔ Some of the design patterns used in Hibernate Framework are:
- ➔ Domain Model Pattern – An object model of the domain that incorporates both behavior and data.
- ➔ Data Mapper – A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.
- ➔ Proxy Pattern for lazy loading-Proxy Design pattern is one of the Structural design pattern and in my opinion one of the simplest pattern to

understand. Proxy pattern intent according to GoF is:

➔ Provide a surrogate or placeholder for another object to control access to it.

➔ The definition itself is very clear and proxy pattern is used when we want to provide controlled access of a functionality. Let's say we have a class that can run some command on the system. Now if we are using it, its fine but if we want to give this program to a client application, it can have severe issues because client program can issue command to delete some system files or change some settings that you don't want.

➔ Factory pattern in SessionFactory-Factory Pattern is one of the Creational Design pattern and it's widely used in JDK as well as frameworks like Spring and Struts.

➔ Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern take out the responsibility of instantiation of a class from client program to the factory class.

What are best practices to follow with Hibernate framework?

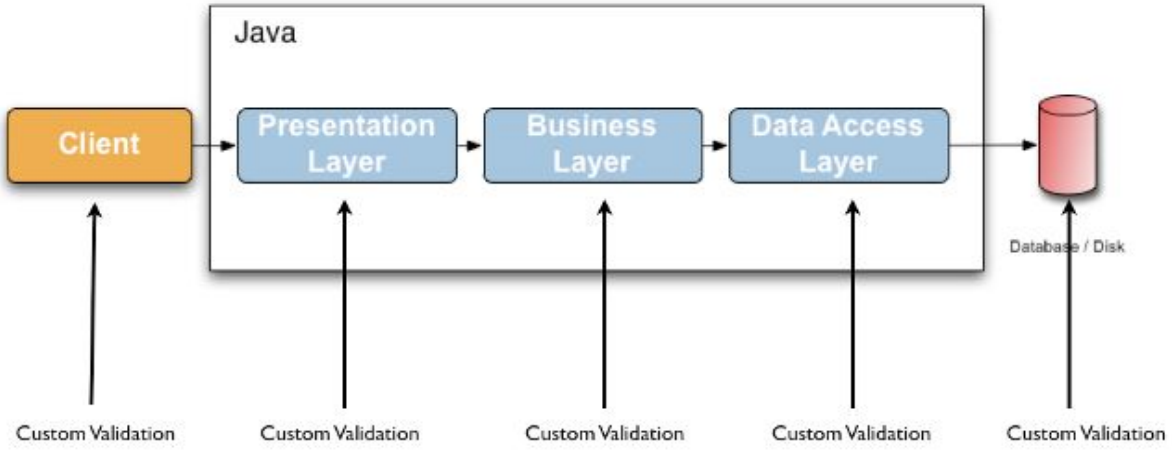
Some of the best practices to follow in Hibernate are :-

- Always check the primary key field access, if it's generated at the database layer then you should not have a setter for this.
- By default hibernate set the field values directly, without using setters. So if you want hibernate to use setters, then make sure proper access is defined as `@Access(value=AccessType.PROPERTY)`.
- If access type is property, make sure annotations are used with getter methods and not setter methods. Avoid mixing of using annotations on both filed and getter methods.
- Use native sql query only when it can't be done using HQL, such as using database specific feature.
- If you have to sort the collection, use ordered list rather than sorting it using Collection API.
- Use named queries wisely, keep it at a single place for easy debugging. Use them for commonly used queries only. For entity specific query, you can keep them in the entity bean itself.
- For web applications, always try to use JNDI DataSource rather than configuring to create connection in hibernate.
- Avoid Many-to-Many relationships, it can be easily implemented using bidirectional One-to-Many and Many-to-One relationships.
- For collections, try to use Lists, maps and sets. Avoid array because you don't get benefit of lazy loading.
- Do not treat exceptions as recoverable, roll back the Transaction and close the Session. If you do not do this, Hibernate cannot guarantee that in-memory state accurately represents the persistent state.
- Prefer DAO pattern for exposing the different methods that can be used

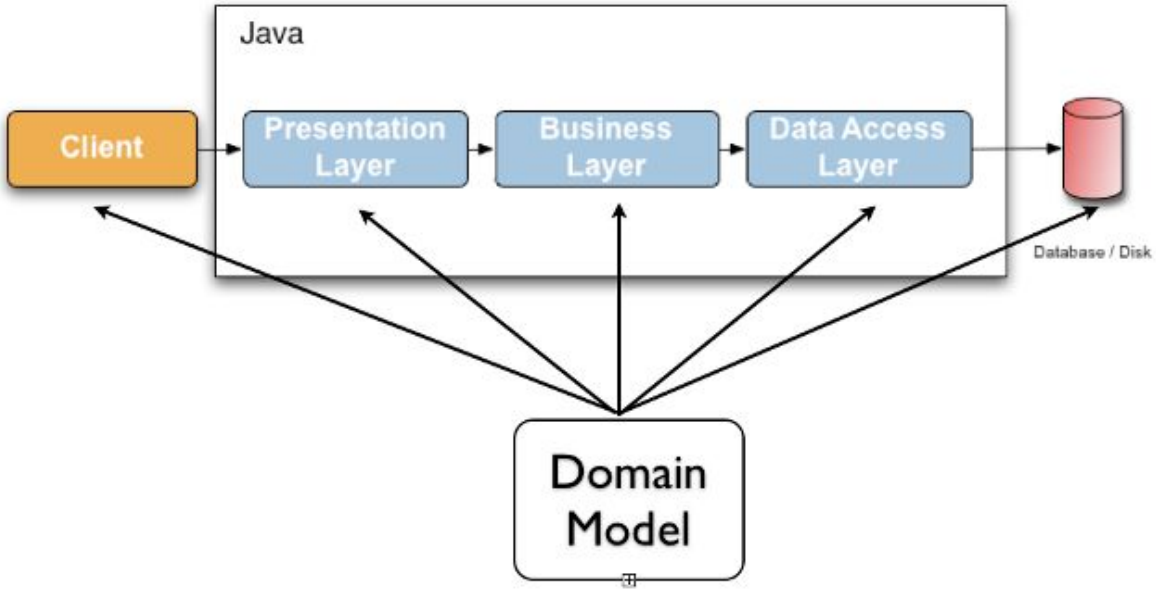
with entity bean
- Prefer lazy fetching for associations

What is Hibernate Validator Framework?

➔ Validating data is a common task that occurs throughout all application layers, from the presentation to the persistence layer. Often the same validation logic is implemented in each layer which is time consuming and error-prone. To avoid duplication of these validations, developers often bundle validation logic directly into the domain model, cluttering domain classes with validation code which is really metadata about the class itself.

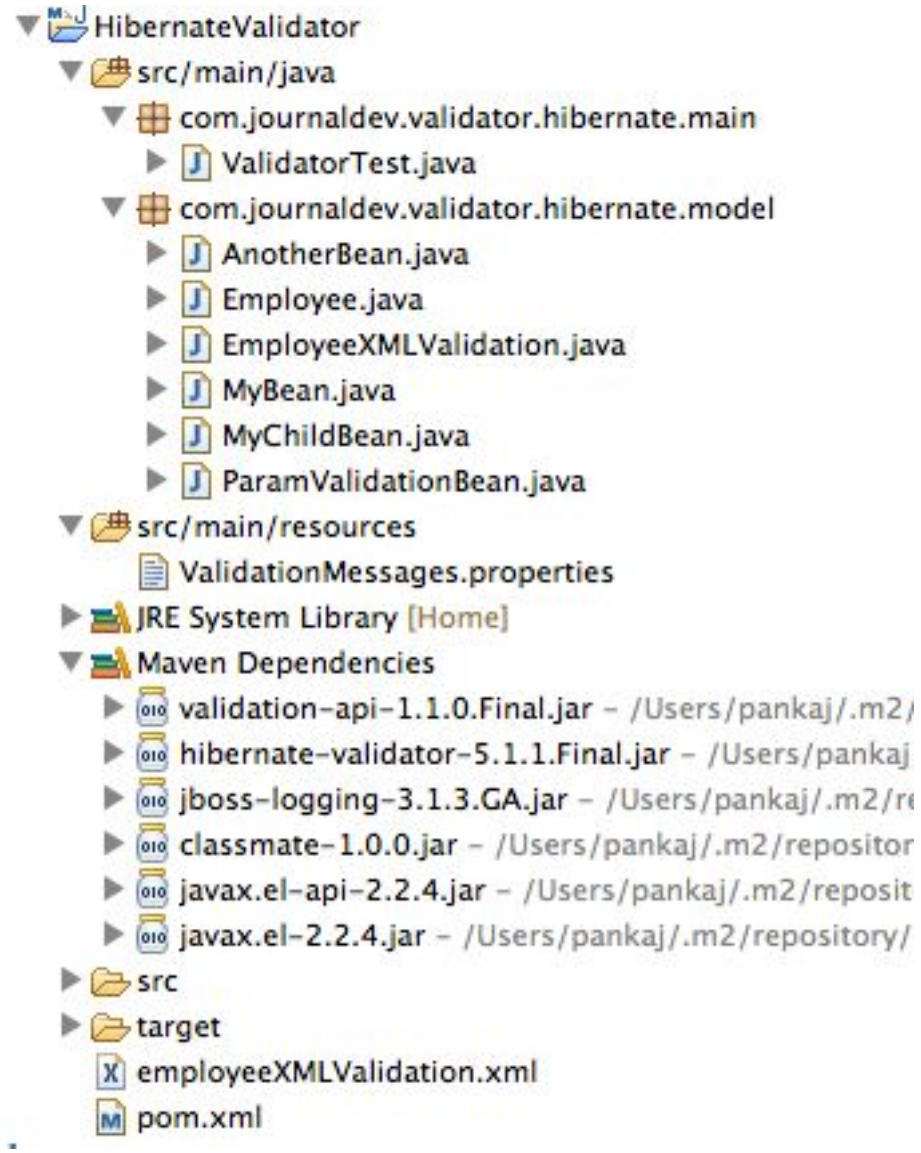


JSR 349 - Bean Validation 1.1 - defines a metadata model and API for entity and method validation. The default metadata source are annotations, with the ability to override and extend the meta-data through the use of XML. The API is not tied to a specific application tier nor programming model. It is specifically not tied to either web or persistence tier, and is available for both server-side application programming, as well as rich client Swing application developers.



Hibernate Validator is the reference implementation of this JSR 349. The implementation itself as well as the Bean Validation API and TCK are all provided and distributed under the Apache Software License 2.0.

Validation is a cross cutting task, so we should try to keep it apart from our business logic. That's why JSR303 and JSR349 provides specification for validating a bean by using annotations. **Hibernate Validator** provides the reference implementation of both these bean validation specs.
It's very easy to use Hibernate Validator and best part is that we can easily extend it and create our own custom validation annotations. Today we will look into the hibernate validator in detail with examples. Finally we will have a test program to check out the validations.
I have created a sample project for all the hibernate validation example, below image shows the final project structure.



Hibernate Validator Maven Dependencies

We need to add JSR303 and Hibernate Validator dependencies to use them.

```
<dependency>
	<groupId>javax.validation</groupId>
	<artifactId>validation-api</artifactId>
	<version>1.1.0.Final</version>
</dependency>
<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-validator</artifactId>
	<version>5.1.1.Final</version>
</dependency>
```

Hibernate Validator also requires an implementation of the Unified Expression Language (JSR 341) for evaluating dynamic expressions in constraint violation messages.

If your application is running in a servlet container such as JBoss, it's already provided. But if you are using it in a standalone application like my example project, you need to add them manually. Required dependencies are;

```
<dependency>
	<groupId>javax.el</groupId>
	<artifactId>javax.el-api</artifactId>
	<version>2.2.4</version>
</dependency>
<dependency>
	<groupId>org.glassfish.web</groupId>
	<artifactId>javax.el</artifactId>
	<version>2.2.4</version>
</dependency>
```

Check the image for all the maven dependencies in the project. Now we are ready to get started with hibernate validation examples.

Hibernate Validation Example

Let's create a simple class and add some validations to it.
Employee.java

```
package com.journaldev.validator.hibernate.model;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.CreditCardNumber;
import org.hibernate.validator.constraints.Email;

public class Employee {

	@Min(value=1, groups=EmpIdCheck.class)
	private int id;

	@NotNull(message="Name cannot be null")
	@Size(min=5, max=30)
	private String name;

	@Email
	private String email;

	@CreditCardNumber
	private String creditCardNumber;

	//default no-args constructor
	public Employee(){}
```

```
	public Employee(int id, String name, String email, String ccNum){
		this.id=id;
		this.name=name;
		this.email=email;
		this.creditCardNumber=ccNum;
	}

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}

	public String getCreditCardNumber() {
		return creditCardNumber;
	}

	public void setCreditCardNumber(String creditCardNumber) {
		this.creditCardNumber = creditCardNumber;
	}

}
```

We should avoid using implementation specific annotations for loose coupling. But hibernate validator provides some very good validation annotations such as @Email and @CreditCardNumber.

We can also provide custom error message to use with any validation. If there are no message defined then hibernate built-in error message will

be used.
We can assign groups to any validation, this can be useful to validate a set of fields only. For example, if I just need to validate the Employee id field, I can use EmpIdCheck group. For this we need to define a class/interface.
EmpIdCheck.java
```
package com.journaldev.validator.hibernate.model;

public interface EmpIdCheck {
}
```
We will look it's usage in the test program later on.

Hibernate Validator Custom Error Messages

We can define our custom error messages too, all we need is to have ValidationMessages.properties file in the classpath.
ValidationMessages.properties
```
#Hibernate Bug for @CreditCardNumber Workaround - https://hibernate.atlassian.net/browse/HV-881
org.hibernate.validator.constraints.LuhnCheck.message=The check digit for ${validatedValue} is invalid, Luhn Modulo 10 checksum failed.

org.hibernate.validator.constraints.Email.message=Invalid email address
```
These property files also support localization, in that case we need to keep file names like ValidationMessages_tr_TR.properties
The property name for message is fully classified annotation name with message at the end, you can easily figure out from above examples.

XML Based Constraints Validation

Sometimes we want validation on third party classes, then we can't use annotations with them. In this situation, xml configuration based validation comes handy. For example, let's say we have a class without any validation annotations like below.
EmployeeXMLValidation.java
```
package com.journaldev.validator.hibernate.model;

public class EmployeeXMLValidation {

	private int id;

	private String name;

	private String email;

	private String creditCardNumber;

	//default no-args constructor
```

```
	public EmployeeXMLValidation(){}

	public EmployeeXMLValidation(int id, String name, String email,
String ccNum){
		this.id=id;
		this.name=name;
		this.email=email;
		this.creditCardNumber=ccNum;
	}

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}

	public String getCreditCardNumber() {
		return creditCardNumber;
	}

	public void setCreditCardNumber(String creditCardNumber) {
		this.creditCardNumber = creditCardNumber;
	}

}
```

A simple example for above bean hibernate validation could be like below.

employeeXMLValidation.xml

```
<constraint-mappings
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/mapping
validation-mapping-1.1.xsd"
	xmlns="http://jboss.org/xml/ns/javax/validation/mapping"
version="1.1">


<default-package>com.journaldev.validator.hibernate.model</default-pa
ckage>

	<bean class="EmployeeXMLValidation" ignore-annotations="true">
		<field name="id">
			<constraint
annotation="javax.validation.constraints.Min">
				<element name="value">1</element>
			</constraint>
		</field>
		<field name="name">
			<constraint
annotation="javax.validation.constraints.NotNull" />
			<constraint
annotation="javax.validation.constraints.Size">
				<element name="min">5</element>
				<element name="max">30</element>
			</constraint>
		</field>
		<field name="email">
			<constraint
annotation="org.hibernate.validator.constraints.Email" />
		</field>
		<field name="creditCardNumber">
			<constraint
annotation="org.hibernate.validator.constraints.CreditCardNumber" />
		</field>
	</bean>

</constraint-mappings>
```

default-package is used to define the base package, so that we can skip long package names.

ignore-annotations is used to tell Hibernate validator to ignore any annotations present in the class for validation purpose, only perform validations as configured in the xml file.

We can have multiple bean validations in a single file, we need to load this file to validator factory configuration, an example of this will be given later on.

Property level constraints

We can apply constraints on the getter methods too, we should not apply it on setter method. Also we should avoid applying constraints on both fields and getter method because then it will be validated twice.

MyBean.java

```
package com.journaldev.validator.hibernate.model;

import javax.validation.constraints.NotNull;

public class MyBean {

	private String name;

	@NotNull
	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}
```

Hibernate Validation with Inheritance

Bean validations are inherited, so if we are validating object of child class then any validations in parent class will also be executed.

MyChildBean.java

```
package com.journaldev.validator.hibernate.model;

import javax.validation.constraints.NotNull;

public class MyChildBean extends MyBean {

	private String data;

	@NotNull
	public String getData() {
		return data;
	}

	public void setData(String data) {
		this.data = data;
	}
}
```

If we will validate instance of MyChildBean, MyBean name field will also be

validated.

Validation with Composition using @Valid annotation

We can use @Valid annotation incase of composition, so that it's validations are executed.

AnotherBean.java

```
package com.journaldev.validator.hibernate.model;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class AnotherBean {

	@NotNull
	@Valid
	private MyChildBean childBean;

	public MyChildBean getChildBean() {
		return childBean;
	}

	public void setChildBean(MyChildBean childBean) {
		this.childBean = childBean;
	}
}
```

Now if we validate AnotherBean instance, MyChildBean object will also be validated.

Method Parameter Hibernate Validation

We can define constraints for method parameters too, a simple example is given below.

ParamValidationBean.java

```
package com.journaldev.validator.hibernate.model;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class ParamValidationBean {

	private String name;

	//using @NotNull at constructor rather than at field
	public ParamValidationBean(@NotNull String name){
		this.name = name;
	}
```

```
	public void printData(@NotNull @Size(min=5) String data){
		System.out.println("Data is::"+data);
	}

	public String getName() {
		return name;
	}

}
```

Hibernate Validator Example Test Program

We have seen a lot of validation scenarios, here is the test program to show the process to validate them.

ValidatorTest.java

```
package com.journaldev.validator.hibernate.main;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.lang.reflect.Method;
import java.util.Set;

import javax.validation.Configuration;
import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
import javax.validation.executable.ExecutableValidator;

import org.hibernate.validator.HibernateValidator;

import com.journaldev.validator.hibernate.model.AnotherBean;
import com.journaldev.validator.hibernate.model.EmpIdCheck;
import com.journaldev.validator.hibernate.model.Employee;
import com.journaldev.validator.hibernate.model.EmployeeXMLValidation;
import com.journaldev.validator.hibernate.model.MyChildBean;
import com.journaldev.validator.hibernate.model.ParamValidationBean;

public class ValidatorTest {

	public static void main(String[] args) throws
FileNotFoundException, NoSuchMethodException, SecurityException {

		//Getting Validator instance with Annotations
		ValidatorFactory validatorFactory =
Validation.buildDefaultValidatorFactory();
		Validator validator = validatorFactory.getValidator();
```

```
		//If there are multiple JSR303 implementations in classpath
		//we can get HibernateValidator specifically too
		ValidatorFactory hibernateVF =
Validation.byProvider(HibernateValidator.class)

.configure().buildValidatorFactory();

		System.out.println("\nSimple field level validation example");
		Employee emp = new Employee(-1, "Name","email","123");
		Set<ConstraintViolation<Employee>> validationErrors =
validator.validate(emp);

		if(!validationErrors.isEmpty()){
			for(ConstraintViolation<Employee> error :
validationErrors){

System.out.println(error.getMessageTemplate()+"::"+error.getPropertyPa
th()+"::"+error.getMessage());

			}
		}

		System.out.println("\nXML Based validation example");

		//XML Based validation
		Configuration<?> config =
Validation.byDefaultProvider().configure();
		config.addMapping(new
FileInputStream("employeeXMLValidation.xml"));
		ValidatorFactory validatorFactory1 =
config.buildValidatorFactory();
		Validator validator1 = validatorFactory1.getValidator();

		EmployeeXMLValidation emp1 = new
EmployeeXMLValidation(10, "Name","email","123");

		Set<ConstraintViolation<EmployeeXMLValidation>>
validationErrors1 = validator1.validate(emp1);

		if(!validationErrors1.isEmpty()){
			for(ConstraintViolation<EmployeeXMLValidation> error
: validationErrors1){

System.out.println(error.getMessageTemplate()+"::"+error.getInvalidValu
e()+"::"+ error.getPropertyPath()+"::"+error.getMessage());
```

```
                }
            }

            System.out.println("\nValidation Group example");
            validationErrors = validator.validate(emp,
EmpIdCheck.class);

            if(!validationErrors.isEmpty()){
                for(ConstraintViolation<Employee> error :
validationErrors){

System.out.println(error.getMessageTemplate()+"::"+error.getPropertyPa
th()+"::"+error.getMessage());

                }
            }

            System.out.println("\nValidation with inheritance example");

            //Validation inheritance and Property-level constraints
example
            MyChildBean childBean = new MyChildBean();
            Set<ConstraintViolation<MyChildBean>>
validationInheritanceErrors = validator.validate(childBean);

            if(!validationInheritanceErrors.isEmpty()){
                for(ConstraintViolation<MyChildBean> error :
validationInheritanceErrors){

System.out.println(error.getMessageTemplate()+"::"+error.getPropertyPa
th()+"::"+error.getMessage());

                }
            }

            System.out.println("\nValidation in composition using @Valid
annotation");

            //@Valid annotation - validation in composition example
            AnotherBean compositionBean = new AnotherBean();
            compositionBean.setChildBean(new MyChildBean());
            Set<ConstraintViolation<AnotherBean>>
validationCompositionErrors = validator.validate(compositionBean);

            if(!validationCompositionErrors.isEmpty()){
```

```
                    for(ConstraintViolation<AnotherBean> error :
validationCompositionErrors){

System.out.println(error.getMessageTemplate()+"::"+error.getPropertyPa
th()+"::"+error.getMessage());

                    }
            }

            System.out.println("\nParameter validation example");
            ParamValidationBean paramValidationBean = new
ParamValidationBean("Pankaj");
            Method method =
ParamValidationBean.class.getMethod("printData", String.class);
            Object[] params = {"1234"};
            ExecutableValidator executableValidator =
validator.forExecutables();
            Set<ConstraintViolation<ParamValidationBean>> violations
=

executableValidator.validateParameters(paramValidationBean, method,
params);
            if(!violations.isEmpty()){
                    for(ConstraintViolation<ParamValidationBean> error :
violations){

System.out.println(error.getMessageTemplate()+"::"+error.getPropertyPa
th()+"::"+error.getMessage());

                    }
            }

      }

}
```

When we run above hibernate validator example program, we get following output.

```
Jul 25, 2014 3:51:56 AM org.hibernate.validator.internal.util.Version
<clinit>
INFO: HV000001: Hibernate Validator 5.1.1.Final

Simple field level validation example
{javax.validation.constraints.Size.message}::name::size must be
between 5 and 30
{org.hibernate.validator.constraints.CreditCardNumber.message}::creditC
ardNumber::invalid credit card number
```

{org.hibernate.validator.constraints.Email.message}::email::Invalid email address

XML Based validation example
{org.hibernate.validator.constraints.Email.message}::email::email::Invalid email address
{javax.validation.constraints.Size.message}::Name::name::size must be between 5 and 30
{org.hibernate.validator.constraints.CreditCardNumber.message}::123::creditCardNumber::invalid credit card number

Validation Group example
{javax.validation.constraints.Min.message}::id::must be greater than or equal to 1

Validation with inheritance example
{javax.validation.constraints.NotNull.message}::data::may not be null
{javax.validation.constraints.NotNull.message}::name::may not be null

Validation in composition using @Valid annotation
{javax.validation.constraints.NotNull.message}::childBean.data::may not be null
{javax.validation.constraints.NotNull.message}::childBean.name::may not be null

Parameter validation example
{javax.validation.constraints.Size.message}::printData.arg0::size must be between 5 and 2147483647

Important points from above hibernate validator test program are:

1. **Validator** instance is thread safe, so we can cache it and reuse it.
2. If there are multiple JSR303 implementation present, then we can get the Hibernate Validator instance using Validation.byProvider() method.
3. Notice the use of validation of a group, it's validating only the fields that are part of the group.
4. **ExecutableValidator** is used to validate the parameters of a method.
5. ExecutableValidator provide methods to validate constructor parameters and return values too, these methods are validateReturnValue(), validateConstructorParameters() and validateConstructorReturnValue()