Contents

[Draw Backs of JDBC: 2](#_Toc171675909)

[What is Hibernate: 3](#_Toc171675910)

[Mapping: 3](#_Toc171675911)

[Configuration: 5](#_Toc171675912)

[**Advantages of hibernates:** 6](#_Toc171675913)

[**Disadvantages of hibernates:** 7](#_Toc171675914)

[**POJO Class:** 8](#_Toc171675915)

[Mapping xml For POJO 10](#_Toc171675916)

[Configuration XML 10](#_Toc171675917)

[How To Install Hibernate 10](#_Toc171675918)

[Follow The Steps: 10](#_Toc171675919)

[Approach 1 13](#_Toc171675920)

[Approach 2: 14](#_Toc171675921)

[Hibernate Versioning 15](#_Toc171675922)

[Importance Of Wrapper And Primitive Types In Hibernate 16](#_Toc171675923)

[Hibernate Lifecycle Of pojo Class Objects 16](#_Toc171675924)

[Transient: 17](#_Toc171675925)

[Persistent: 17](#_Toc171675926)

[Detached: 17](#_Toc171675927)

[Inheritance Mapping In Hibernate – Introduction 18](#_Toc171675928)

[Hibernate Inheritance: Table Per Class Hierarchy 19](#_Toc171675929)

[Hibernate Inheritance: Table Per subClass Hierarchy 19](#_Toc171675930)

[Hibernate Inheritance: Table Per Concrete Class Hierarchy 20](#_Toc171675931)

[Example On Composite Primary Keys In Hibernate 21](#_Toc171675932)

[Generators <generator> In Hibernate 22](#_Toc171675933)

[List of generators 22](#_Toc171675934)

[1) assigned 22](#_Toc171675935)

[2) increment 23](#_Toc171675936)

[3) sequence 23](#_Toc171675937)

[4) hilo 24](#_Toc171675938)

[5) native 24](#_Toc171675939)

[6) identity 25](#_Toc171675940)

[7) seqhilo 25](#_Toc171675941)

[8) uuid 25](#_Toc171675942)

[9) guid 25](#_Toc171675943)

[10) select 25](#_Toc171675944)

[11) foreign 25](#_Toc171675945)

[12) sequence-identity 25](#_Toc171675946)

[Hibernate Query Language 25](#_Toc171675947)

[Advantages Of HQL: 26](#_Toc171675948)

[Procedure To Execute HQL Command: 27](#_Toc171675949)

[Different Ways Of Executing HQL 28](#_Toc171675950)

[**Case 1: [ Selecting Complete Object ]** 28](#_Toc171675951)

[**Case 2: [ Selecting Partial Object ]** 29](#_Toc171675952)

[**Case 3: [ Selecting Partial Object ]** 29](#_Toc171675953)

[Criteria Query 29](#_Toc171675954)

[Adding Conditions To Criteria 30](#_Toc171675955)

[Projections 31](#_Toc171675956)

[Difference between HQL and Criteria Query in Hibernate 33](#_Toc171675957)

[Advantages and Disadvantages of Native SQL 33](#_Toc171675958)

# Draw Backs of JDBC:

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

# JPA Vs Hibernate

## JPA

* JPA is only a specification, it is not an implementation.
* It is a set of rules and guidelines to set interfaces for implementing object-relational mapping, .
* It needs a few classes and interfaces.
* It supports simple, cleaner, and assimilated object-relational mapping.
* It supports polymorphism and inheritance.
* Dynamic and named queries can be included in JPA.
* Hibernate is an implementation of JPA guidelines.
* It helps in mapping Java data types to SQL data types.
* It is the contributor of JPA.

JPA is a specification. It gives common functionality and prototype to ORM tools. All ORM tools (such as Hibernate) follow the common standards, by executing the same specification. Subsequently, if we need to switch our application from one ORM tool to another then we can easily do it.

*The major difference between Hibernate and JPA is that Hibernate is a framework while JPA is API specifications. Hibernate is the implementation of all the JPA guidelines*

# What is Hibernate:

Hibernate is the ORM tool given to transfer the data between a java (object) application and a database (Relational) in the form of the objects.

Hibernate is a non-invasive framework,  means it wont forces the programmers to extend/implement any class/interface, and in hibernate we have all POJO classes so its light weight.’ Hibernate can runs with in or with out server, i mean it will suitable for all types of java applications (stand alone or desktop or any servlets bla bla.)

Hibernate is purely for persistence (to store/retrieve data from Database).

Mapping and Configuration are very familiar keywords we used to here in the hibernate, every hibernate program must need these 2 xml files.

## **Mapping:**

* Mapping file is the heart of hibernate application.
* Every ORM tool needs this mapping, mapping is the mechanism of placing an object properties into column’s of a table.
* Mapping can be given to an ORM tool either in the form of an XML or in the form of the annotations.
* The mapping file contains mapping from a pojo class name to a table name and pojo class variable names to table column names.
* While writing an hibernate application, we can construct one or more mapping files, mean a hibernate application can contain any number of  mapping files.

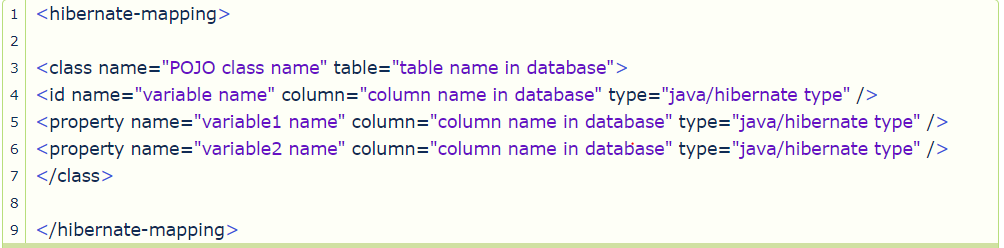
generally an object contains 3 properties like

* Identity (Object Name)
* State (Object values)
* Behavior (Object Methods)

But while storing an object into the database, we need to store only the values(State) right ? but how to avoid identity, behavior.. its not possible. In order to inform what value of an object has to be stored in what column of the table, will be taking care by the mapping,  actually mapping can be done using 2 ways,

* XML
* Annotations.

Actually annotations are introduced into java from JDK 1.5.



## Configuration:

Configuration is the file loaded into an hibernate application when working with hibernate, this configuration file contains 3 types of information..

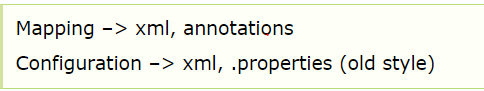
* Connection Properties
* Hibernate Properties
* Mapping file name(s)

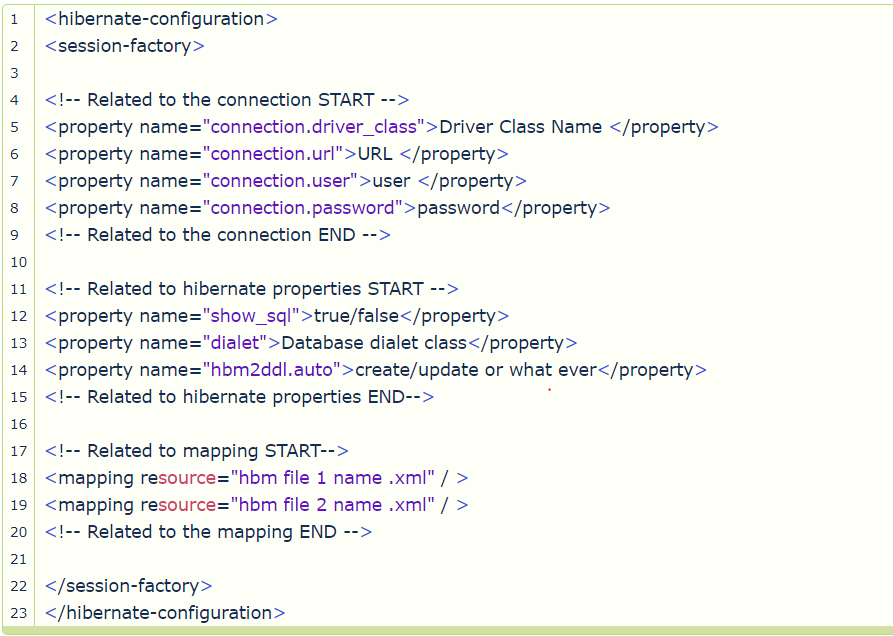
We must create one configuration file for each database we are going to use, suppose if we want to connect with 2 databases, like Oracle, MySql, then we must create 2 configuration files.

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

We can write this configuration in 2 ways…

* xml
* By writing Properties file.  We don’t have annotations here, actually in hibernate 1, 2.x we defined this configuration file by writing .properties file, but from 3.x xml came into picture.





**Advantages of hibernates:**

* Hibernate supports Inheritance, Associations, Collections
* In hibernate if we save the derived class object,  then its base class object will also be stored into the database, it means hibernate supporting inheritance
* Hibernate supports relationships like One-To-Many,One-To-One, Many-To-Many-to-Many, Many-To-One
* This will also supports collections like List,Set,Map (Only new collections)
* In jdbc all exceptions are checked exceptions, so we must write code in try, catch and throws, but in hibernate we only have Un-checked exceptions, so no need to write try, catch, or no need to write throws.  Actually in hibernate we have the translator which converts checked to Un-checked
* Hibernate has capability to generate primary keys automatically while we are storing the records into database
* Hibernate has its own query language, i.e hibernate query language which is database independent
* So if we change the database, then also our application will works as HQL is database independent
* HQL contains database independent commands
* While we are inserting any record, if we don’t have any particular table in the database, JDBC will rises an error like “View not exist”, and throws exception, but in case of hibernate, if it not found any table in the database this will create the table for us
* Hibernate supports caching mechanism by this, the number of round trips between an application and the database will be reduced, by using this caching technique an application performance will be increased automatically.
* Hibernate supports annotations, apart from XML
* Hibernate provided Dialect classes, so we no need to write sql queries in hibernate, instead we use the methods provided by that API.
* Getting pagination in hibernate is quite simple.

**Disadvantages of hibernates:**

* I don’t think there are disadvantages in hibernate
* You know some thing.., Its saying hibernate is little slower than pure JDBC, actually the reason being hibernate used to generate many SQL statements in run time, but i guess this is not the disadvantage
* But there is one major disadvantage, which was boilerplate code issue, actually we need to write same code in several files in the same application, but spring eliminated this

Any hibernate application, for example consider even first hello world program must always contains 4 files totally.

* POJO class
* Mapping XML
* Configuration XML
* One java file to write our logic

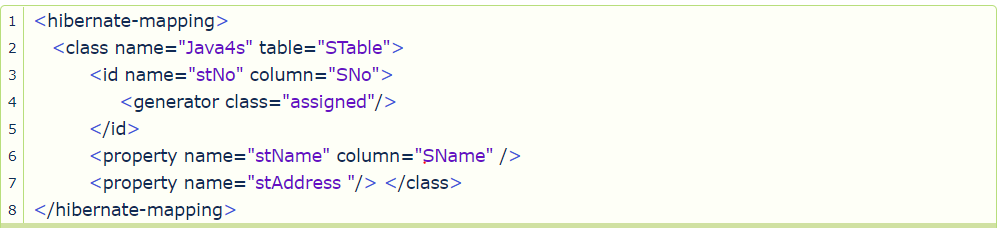
Actually these are the minimum requirement to run any hibernate application, and in fact we may require any number of POJO classes and any number of mapping xml files (**Number of POJO classes = that many number of mapping xmls**), and only one configuration xml and finally one java file to write our logic.

**POJO Class:**

* POJO is a simple java file, no need to extend any class or implement any interface.
* This POJO class contain private properties variables, and for each property a setter and a getter



## **Mapping xml For POJO**



Yes., see in this above mapping xml, for stAddress property i have not written any column name i just been specified  **<property name=”stAddress “/>,**this means in the database the column name for stAddress property will also be stAddress, in these cases we can ignore the column attribute to write, and i will explain about this <generator /> element later.

## Configuration XML

Refer above

Usually configuration file name will be hibernate.cfg.xml

# How To Install Hibernate

Download hibernate.jar and use

For maven :

<dependency>

<groupId>org.hibernate</groupId>

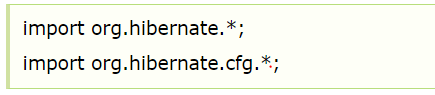
<artifactId>hibernate-core</artifactId>

<version>5.4.0.Final</version>

</dependency>

## Follow The Steps:

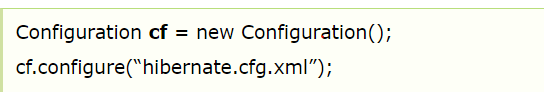
**1.** Import the hibernate API, they are many more, but these 2 are more than enough…



**2.** Among Configuration, Mapping xml files, first we need to load configuration xml, because once we load the configuration file, automatically mapping file will be loaded as we registered this mapping xml in the configuration file.

So to load configuration xml, we need to create object of **Configuration** class, which is given in **org.hibernate.cfg.\***;  and we need to call **configure()** method in that class, by passing xml configuration file name as parameter.

Eg:



Here our configuration file name is your choice, but by default am have been given hibernate.cfg.xml,  so once this configuration file is loaded in our java app, then we can say that hibernate environment is started in our program.

So once we write the line\_ **cf.configure(“hibernate.cfg.xml”),**configuration object**cf**will reads this xml file**hibernate.cfg.xml,**actually internally cf will uses DOM parsers to read the file.

Finally…

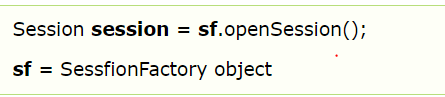
* cf will reads data from hibernate.cfg.xml
* Stores the data in different variables
* And finally all these variables are grouped and create one high level hibernate object we can call as SessionFactory object.
* So Configuration class only can create this SessionFactory object



Actually SessionFactory is an interface not a class, and SessionFactoryImpl is the implimented class for SessionFactory, so we are internally creating object of SessionFactoryImpl class and storing in the interface reference, so this SessionFactory object **sf**contains all the data regarding the configuation file so we can call**sf**as heavy weight object.

3. Creating an object of session,

* Session is an interface and SessionImpl is implemented class, both are given in org.hibernate.\*;
* When ever session is opened then internally a database connection will be opened, in order to get a session or open a session we need to call openSession() method in SessionFactory, it means SessionFactory produces sessions.



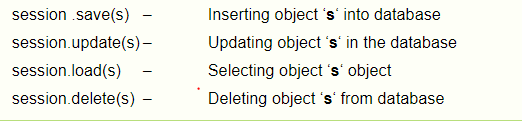
4. Create a logical transaction

While working with insert, update, delete, operations from an hibernate application onto the database then hibernate needs a logical Transaction, if we are selecting an object from the database then we do not require any logical transaction in hibernate.  In order to begin a logical transaction in hibernate then we need to call a method beginTransaction() given by Session Interface.

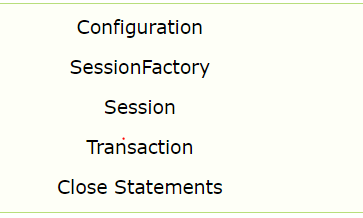
Transaction tx = **sessio**n.beginTransaction();

**session** is the object of Session Interface

5. Use the methods given by Session Interface,  to move the objects from application to database and  from database to application



* So finally we need to call **commit()**in Transaction, like **tx.commit()**;
* As i told earlier,  when we open session a connection to the database will be created right, so we must close that connection as session. close().
* And finally close the SessionFactory as **sf.close()**



Select , delete ,insert code available in project.

**Update**

this is the program to update an object (1 complete row) in the database, which is already persisted in the database, then we have the following two approaches

## Approach 1

Load that object from the database, and modify its values, now hibernate automatically modifies the values on to database also, when ever the transaction is committed.

* When ever an object is loaded from the database then hibernate stores the loaded object in cache-memory maintained by session-interface
* Once an object is loaded, if we do any modifications on that object by calling its setter methods, then these modification are stored in the object maintained by cache-memory
* if we modify the loaded object for multiple times then also the modifications will be stored in object maintained by the cache-memory only.
* when ever we issue commit() operation then hibernate verify whether any changes are there between the object stored in the cache and object in the database, if changes exists then hibernate automatically updates the database by generating any update operation.
* What am saying is hibernate automatically maintains synchronization between cache-memory object and database table objects (rows)

## Approach 2:

If we want to modify object in the database, then create new object with same id and we must call update() given by session interface.

first approach is  recommended always.

First new configuration() is called this will initialize everything. Namedquires, classes, cache.

New configuration().configure() will read the hibernate configuration file(hibernate.cfg.xml). and reads all the properties defined in the configuration file.

new Configuration().configure().buildSessionFactory(); will create the sessionfactory.

Most applications create a Hibernate SessionFactory singleton that’s cached for the lifecycle of the app because the object is resource-intensive to create.

While creating session factory. All the JDBC settings will be configured. Like batch size, flush.

Sessionfactory.getsession() will create session. session.beginTransaction() will begin the transaction.

EntiryManagerFactor and sessionfactory both are used for same perpose.

Entitymanager and session are both used for same purpose.

ust like the SessionFactory and EntityManagerFactory, the Hibernate Session extends the JPA EntityManager. So, all methods defined by the EntityManager are available in the Hibernate Session.

<https://www.waitingforcode.com/hibernate/difference-between-sessionfactory-and-entitymanagerfactory/read>

# Hibernate Versioning

once an object is saved in a database, we can modify that object any number of times right, If we want to know how many no of times that an object is modified then we need to apply this versioning concept.

When ever we use versioning then hibernate inserts version number as **zero**, when ever object is saved for the first time in the database.  Later hibernate increments that version no by one automatically when ever a modification is done on that particular object.  
In order to use this versioning concept, we need the following two changes in our application

* Add one property of type int in our pojo class
* In hibernate mapping file, add an element called version soon after id element
* // using version annotation.
* @Version
* private int version;

**Note**:

* Remember friends, first we must run the logic to **save** the object then hibernate will inset 0 (Zero) by default in the version column of the database, its very important point in the interview point of view also
* First save logic to let the hibernate to insert zero in the version column, then any number of update logic’s (programs) we run, hibernate will increments +1 to the previous value
* But if we run the update logic for the first time, hibernate will not insert zero..! it will try to increment the previous value which is NULL in the database so we will get the exception.

Guys your know some thing.., actually we can run any logic (Save or Update) for the first time, but make sure the versioning column is a number (>=0), but save logic has ability to insert zero by default if there is no value, and update logic will directly tries to increments already existing value by 1, it wont insert any value by default if its null

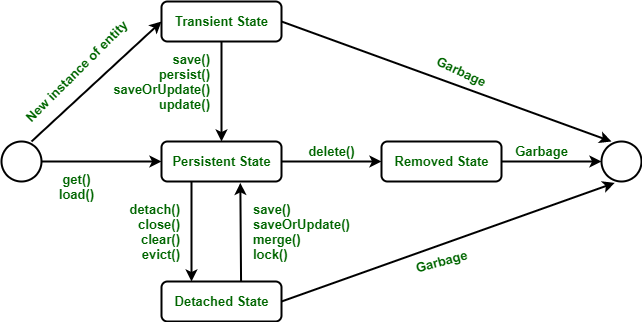
* **CascadeType.ALL** – Propagates all operations from parent to target entity.
* **CascadeType.PERSIST** – Propagates persist from parent to target entity.
* **CascadeType.MERGE** – Propagates merge from parent to target entity.
* **CascadeType.REMOVE** – Propagates remove from parent to target entity.
* **CascadeType.REFRESH** – Propagates refresh from parent to target entity.
* **CascadeType.DETACH** – Propagates detach from parent to target entity.

# Importance Of Wrapper And Primitive Types In Hibernate

If we use primitive types as variables in pojo classes and while inserting if we don’t set values for this then default primitive values will be inserted (ex for int as 0). If we use wrapper classes like Integer then null values will be inserted.

# Hibernate Lifecycle Of pojo Class Objects

<https://www.geeksforgeeks.org/hibernate-lifecycle/?ref=lbp>



Actually our POJO class object having 3 states like

* Transient state
* Persistent state
* Detached state

## Transient:

One newly created object,with out having any relation with the database, means never persistent, not associated with any Session object

## Persistent:

Having the relation with the database, associated with a unique Session object

## Detached:

previously having relation with the database [persistent ], now not associated with any Session

see the next sessions

* When ever an object of a pojo class is created then it will be in the Transient state
* When the object is in a Transient state it doesn’t represent any row of the database, i mean not associated with any Session object, if we speak more we can say no relation with the database its just an normal object
* If we modify the data of a pojo class object, when it is in transient state then it doesn’t effect on the database table
* When the object is in persistent state, then it represent one row of the database, if the object is in persistent state then it is associated with the unique Session
* if we want to move an object from persistent to detached state, we need to do either closing that session or need to clear the cache of the session
* if we want to move an object from persistent state into transient state then we need to delete that object permanently from the database

if we want to convert the object from Transient state to Persistent state we can do in 2 ways

* By saving that object like above
* By loading object from database

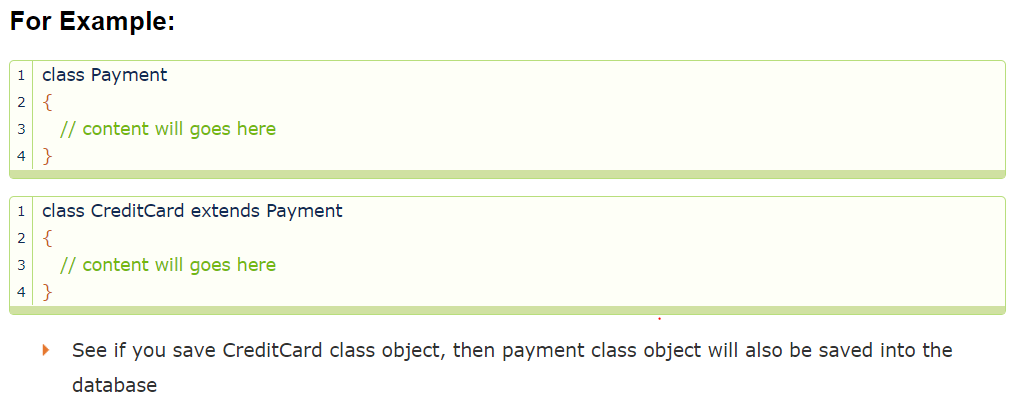
If we do any modifications all the changes will first applied to the object in session cache only (Let\_\_ we do the modifications 5 times, then 5 times we need to save the changes into the database right, which means number of round trips from our application to database will be increased, Actually if we load an object from the database, first it will saves in the cache-memory so if we do any number of changes all will be effected at cache level only and finally we can call save or update method so with the single call of save or update method the data will be saved into the database.

If we want to save an object into database then we need to call any one of the following 3 methods

* save()
* persist()
* saveOrUpdate()

# Inheritance Mapping In Hibernate – Introduction

compared to JDBC we have one main advantage in hibernate, which is hibernate inheritance.  Suppose if we have base and derived classes, now if we save derived(sub) class object, base class object will also be stored into the database.  
But the thing is we must specify in what table we need to save which object data



**Hibernate supports 3 types of Inheritance Mappings**:

* Table per class hierarchy
* Table per sub-class hierarchy
* Table per concrete class hierarchy

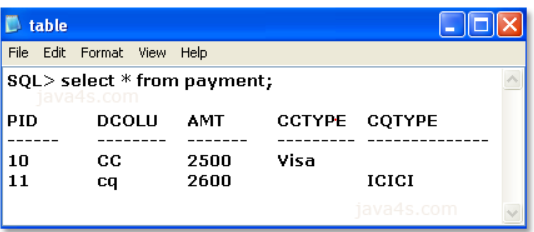
# Hibernate Inheritance: Table Per Class Hierarchy

here is the explanation and one example on hibernate table per class hierarchy, consider we have base class named Payment and 2 derived classes like CreditCard, Cheque



If we save the derived class object like CreditCard or Cheque then automatically Payment class object will also be saved into the database, and in the database all the data will be stored into a **single table** only, which is base class table for sure.

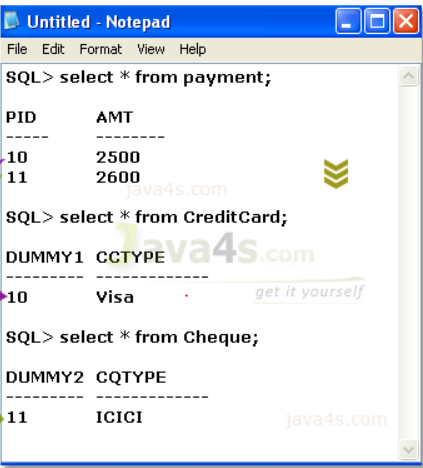
But here we must use one extra discriminator column in the database,  just to identify which derived class object we have been saved in the table along with the base class object,  if we are not using this column hibernate will throws the exception



# Hibernate Inheritance: Table Per subClass Hierarchy

x number of classes = x number of tables in the database

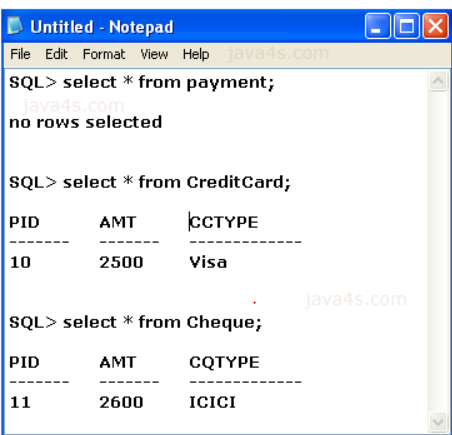
If we save the CreditCard class object, then first hibernate will saves the data related to super class object into the super class related table in the database and then CreditCard object data in CreditCard related table in the database, so first base class data will be saved



# Hibernate Inheritance: Table Per Concrete Class Hierarchy

x number of derived classes = x number of tables in the database

* Once we save the derived class object, then derived class data and base class data will be saved in the derived class related table in the database
* for this type we need the tables for derived classes, but not for the base class
* in the mapping file we need to use one new element <union-subclass — >under <class —>



# Example On Composite Primary Keys In Hibernate

* If the table has a primary key then in the hibernate mapping file we need to configure that column by using <id /> element right..!
* Even though the database table doesn’t have any primary key, we must configure one column as id (one primary key is must)
* If the database table has more than one column as primary key then we call it as composite primary key, so if the table has multiple primary key columns , in order to configure these primary key columns in the hibernate mapping file we need to use one new element called **<composite-id** …..> **</composite-id>**

Actually if we have a single primary key, we need to use <id> element, but this time we have multiple primary keys, so we need to use this new element <composite-id>

# Generators <generator> In Hibernate

The <generator> class is a sub-element of id. It is used to generate the unique identifier for the objects of persistent class. There are many generator classes defined in the Hibernate Framework.

All the generator classes implements the **org.hibernate.id.IdentifierGenerator**[**interface**](https://www.javatpoint.com/interface-in-java). The application programmer may create one's own generator classes by implementing the IdentifierGenerator interface. Hibernate framework provides many built-in generator classes:

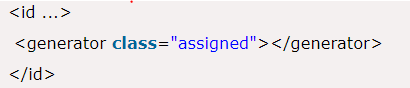
# List of generators

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

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

### 1) assigned

* This generator supports in all the databases
* This is the default generator class used by the hibernate, if we do not specify <generator –> element under id element then hibernate by default assumes it as “assigned”
* If generator class is assigned, then the programmer is responsible for assigning the primary key value to object which is going to save into the database

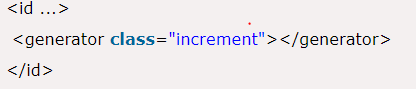


### 2) increment

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

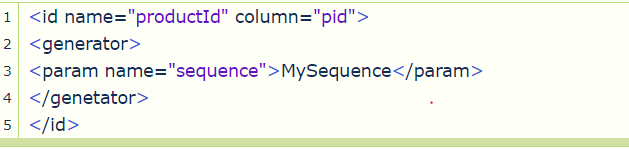
Max of id value in Database + 1

* if we manually assigned the value for primary key for an object, then hibernate doesn’t considers that value and uses **max value of id in database + 1** concept only 🙂
* If there is no record initially in the database, then for the first time this will saves primary key value as 1, as…



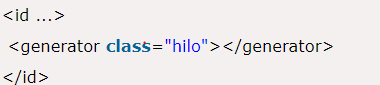
### 3) sequence

* Not has the support with MySql
* This generator class is database dependent it means, we cannot use this generator class for all the database, we should know whether the database supports sequence or not before we are working with it
* while inserting a new record in a database, hibernate gets next value from the sequence under assigns that value for the new record
* If programmer has created a sequence in the database then that sequence name should be passed as the generator



* If the programmer has not passed any sequence name, then hibernate creates its own sequence with name “**Hibernate-Sequence**” and gets next value from that sequence, and than assigns that id value for new record
* But remember, if hibernate want’s to create its own sequence, in hibernate configuration file, **hbm2ddl.auto** property must be set enabled

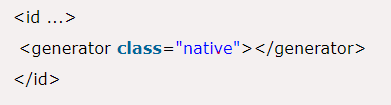
### 4) hilo



### 5) native

when we use this generator class, it first checks whether the database supports identity or not, if not checks for sequence and if not, then hilo will be used finally the order will be..

* identity
* sequence
* hilo



### 6) identity

* This is database dependent, actually its not working in oracle
* In this case (identity generator) the id value is generated by the database, but not by the hibernate, but in case of increment hibernate will take over this
* this identity generator doesn’t needs any parameters to pass
* this identity generator is similar to increment generator, but the difference was increment generator is database independent and hibernate uses a select operation for selecting max of id before inserting new record
* But in case of identity, no select operation will be generated in order to insert an id value for new record by the hibernate

### 7) seqhilo

It uses high and low algorithm on the specified sequence name. The returned id is of type short, int or long.

### 8) uuid

It uses 128-bit UUID algorithm to generate the id. The returned id is of type String, unique within a network (because IP is used). The UUID is represented in hexadecimal digits, 32 in length

### 9) guid

|  |
| --- |
| It uses GUID generated by database of type string. It works on MS SQL Server and MySQL. |

### 10) select

|  |
| --- |
| It uses the primary key returned by the database trigger. |

### 11) foreign

|  |
| --- |
| It uses the id of another associated object, mostly used with <one-to-one> association. |

### 12) sequence-identity

|  |
| --- |
| It uses a special sequence generation strategy. It is supported in Oracle 10g drivers only. Hibernate Query Language so far we done the operations on single object (single row), here we will see modifications,  updates on multiple rows of data (multiple objects) at a time.  In hibernate we can perform  the operations on a single row (or) multiple rows at a time, if we do operations on multiple  rows at once, then we can call this as bulk operations.   * HQL is the own query language of hibernate and it is used to perform bulk operations on hibernate programs * An object oriented form of SQL is called HQL * here we are going to replace table column names  with POJO class variable names and table names with POJO class names in \   order to get HQL commands Advantages Of HQL:  * HQL is database independent, means if we write any program using HQL commands then our program will be able to execute in\    all the databases with out doing any further changes to it   * HQL supports object oriented features like ***Inheritance***, ***polymorphism***, ***Associations***(Relation ships)        Procedure To Execute HQL Command:  * If we want to execute execute an HQL query on a database, we need to create a query object * ” Query ” is an interface given in org.hibernate package * In order to get query object, we need to call createQuery() method in the session Interface * Query is an interface, QueryImpl is the implemented class * we need to call list method for executing an HQL command on database, it returns java.util.List * we need to use java.util.Iterator for iterating the List collection    Different Ways Of Executing HQL**Case 1: [ Selecting Complete Object ]**  **Case 2: [ Selecting Partial Object ]**  **Case 3: [ Selecting Partial Object ]**  Criteria Query unlike HQL, Criteria is only for selecting the data from the database, that to we can select complete objects only not partial objects,  in fact by combining criteria and projections concept we can select partial objects too   Adding Conditions To Criteria  * If we want to put conditions to load data from database, using criteria then we need to create one Criterion Interface object    and we need to add this object to Criteria Class object   * Criterion is an interface given in “org.hibernate.criterion” package * In order to get Criterion object, we need to use Restrictions class * Restrictions is the factory for producing Criterion objects, but friends there is no explicit relation between Criterion interface   and Restrictions class, it means Restrictions class is not implemented from Criterion Interface   * In Restrictions class, we have all static methods and each method of this class returns Criterion object * Restrictions class is also given in “org.hibernate.criterion” package     above example am fetching the data by comparing price greater than (>) 12000   * If we want to put more conditions on the data (multiple conditions) then we can use **and** method ,   **or** method give by the Restrictions class |

if we want to add some sorting order for the objects, before the objects are going to store in list object then we need to add an Order class object to the Criteria class object by calling addOrder() method..,

* Order is a class given in “org.hibernate.Criterion” package
* In Order class, we have 2 static methods, asc()[ascending order] and dsc()[descending order] for getting an objects in required order
* Internal concept is, hibernate will select the records (rows) from PRODUCT table and stores them into a ResultSet and then converts each row data of resultset into a POJO class object basing on our field type, then all these objects into a list according to the order you have given

# Projections

The projections concept is introduced in hibernate 3.0 and mainly we can do the following 2 operations using the projection

* We can load partial object from the database
* We can find the Result of Aggregate functions

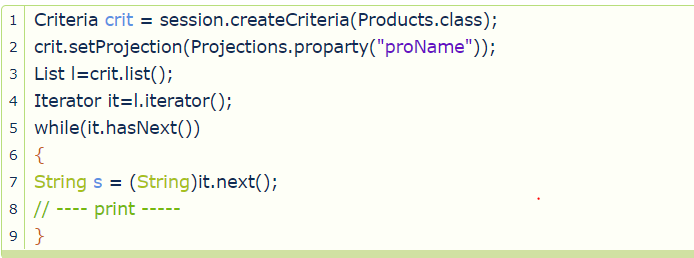
Projection is an Interface given in “org.hibernate.criterion” package, Projections is an class given in same package,  actually Projection is an interface, and Projections is an class and is a factory for producing projection objects.

In Projections class, we have all static methods and each method of this class returns Projection interface object.

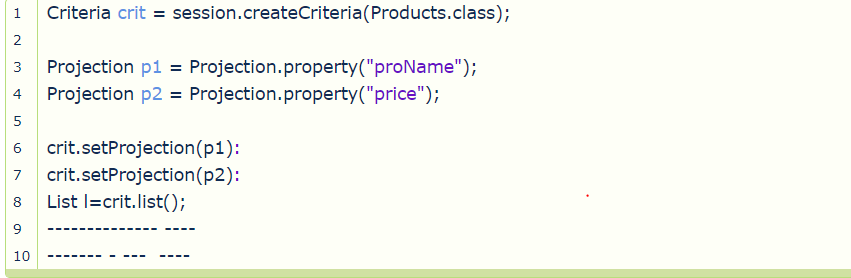
If we want to add a Projection object to Criteria then we need to call a method **setProjection()**

**Remember**, while adding projection object to criteria, it is possible to add one object at a time.  It means if we add 2nd projection object then this 2nd one will overrides the first one (first one wont be work), so at a time we can only one projection object to criteria object.

Using criteria, if we want to load partial object from the database, then we need to create a projection object for property that is to be loaded from the database



If we add multiple projections to criteria then the last projection added will be considered to execute



# Difference between HQL and Criteria Query in Hibernate

* HQL is to perform both select and non-select operations on the data,  but Criteria is only for selecting the data, we cannot perform non-select operations using criteria
* HQL is suitable for executing Static Queries, where as Criteria is suitable for executing Dynamic Queries
* HQL doesn’t support pagination concept, but we can achieve pagination with Criteria
* Criteria used to take more time to execute then HQL
* With Criteria we are safe with SQL Injection because of its dynamic query generation but in HQL as your queries are either fixed or parametrized, there is no safe from SQL Injection.

Native SQL is another technique of performing bulk operations on the data using hibernate

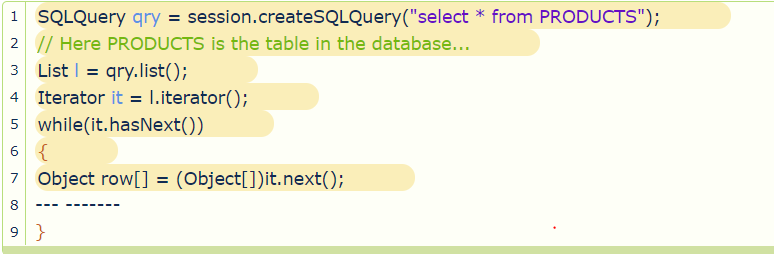
* By using Native SQL, we can perform both select, non-select operations on the data
* In face Native SQL means using the direct SQL command specific to the particular (current using) database and executing it with using hibernate

## 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 draw back 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.

**Remember**, while executing Native SQL Queries, even though we are selecting complete objects from teh database we need to type cast into object array only, not into our pojo class type, because we are giving direct table, column names in the Native SQL Querie so it does’nt know our class name

If we execute the command, always first it will put’s data in **ResultSet** and from there List



* while selecting data from the table, even though you are selecting the complete object from the table, in while loop still we type cast into object array only right
* See the above code, we typecast into the object[] arrays right..,  in case if we want to type cast into our POJO class (i mean to get POJO class obj), then we need to go with entityQuery concept
* In order to inform the hibernate that convert each row of ResultSet into an object of the POJO class back, we need to make the query as an entityQuery
* to make the query as an entityQuery, we need to call addEntity() method

i have been added addEntity(Product.class) at the end, which will let the hibernate to know about our POJO class, so now we can typecast into our POJO class type

# Caching

Caching in Hibernate means storing and reusing frequently used data to speed up your application. There are two kinds of caching: Session-level and SessionFactory-level. Level 1 cache is a cache that stores objects that have been queried and persist to the current session. This cache helps to reduce the number of database round trips by storing objects that are frequently accessed in the session. Level 1 cache is the default cache that is stored by default in hibernate. It is related to the hibernate session object.

Hibernate’s Level 2 caching is a built-in caching mechanism that allows you to cache data across several sessions. Level 2 cache stores data in your application’s shared cache region, which is accessible by every Hibernate session in your application. Level 2 caching can help optimize your application by reducing the number of database queries needed to load data. There are several cache providers available for Hibernate. You can use any of them, including Ehcache, Infinispan, Hazelcast, and others.

## Cache Eviction

## <https://www.geeksforgeeks.org/hibernate-cache-eviction-with-example/?ref=lbp>

## The process of cache eviction involves the deletion or replacement of data stored in a cache. This is a critical component of cache management as it guarantees that the cache remains functional and efficient. Here are some reasons why cache eviction is important:

1. **Maximizing cache space:**Caches have a limited amount of space, and if the cache becomes full, new data cannot be stored. Eviction ensures that the cache is not filled with outdated or unused data, allowing space for new data.
2. **Improved cache performance:**Eviction ensures that the cache only contains relevant data, increasing the cache hit rate and improving overall performance.
3. **Avoiding data staleness:**Data in a cache may become stale over time, especially if it is rarely accessed. Eviction removes such data, ensuring that only fresh data is stored in the cache.
4. **Reducing cache access time:**Eviction ensures that the cache only contains relevant data, which reduces the time needed to access data from the cache.
5. **Reducing cache misses:**Eviction helps reduce the occurrence of cache misses, which are situations where requested data is not found in the cache, and must be retrieved from slower storage such as a disk

When a cache gets full, it can’t store new data without getting rid of old data. That’s what cache eviction is all about. It’s the process of getting rid of or swapping out data in the cache to make room for new stuff. It’s done by using algorithms to figure out which data to get rid of. There are different policies for doing this, like LRU (Least Recently Used), LFU (Frequently Used), and RPP (Random Replacement). These policies are picked based on the type of data being stored and how often it’s accessed. The idea is to make sure the cache only has stuff that’s important and frequently accessed. That way, it’ll be more efficient and work better.

Cache eviction is the process of removing data from a cache when the cache becomes full or when the data is no longer needed. There are several different types of cache eviction algorithms used by computer systems, including:

1. Time-Based Eviction
2. Count-Based Eviction
3. Query Result Eviction
4. Cache Region Clearing

## Cache Expiration

## https://www.geeksforgeeks.org/hibernate-cache-expiration/?ref=lbp

The process of cache expiration is the setting of a time limit on the retention and utilization of data stored in a cache before it is deemed obsolete and invalid. When data is stored in a cache, it is temporarily stored in memory or disk, to optimize system performance and minimize the burden on the underlying data source. On the other hand, if the data is not regularly updated or removed from the cache, it may become stale and result in inaccurate or irrelevant results. Setting a cache expiration time ensures that the system regularly checks for and removes obsolete data from the cache to ensure that only the most relevant and up-to-date data is used.

## There are mainly two types of caching:

* First level cache
* Second-level cache

## **1). First level cache**

The First level cache is **by default enabled** by Hibernate itself. The session object maintains the first-level cache.

An application can have many sessions. Data hold by one session object is not accessible to the entire application — means the data of a particular session is not shared with other sessions of the application. So you can use the first-level cache to store local data i.e. required by the session itself.

So when you query an entity or object, for the very first time it is retrieved from the database and stored into the first-level cache (associated with the hibernate session). If we query for the same entity or object again with the same session object, it will be loaded from cache and no SQL query will be executed

## Some useful methods:

* **Session.evict():** to remove the cached/stored entity.
* **refresh():** method to refresh the cache.
* **clear():** method to remove all the entities from the cache.

**Note:** You **can not disable** the first-level cache, it is enabled by the hibernate itself. Hibernate entities or database rows remain in cache only until Session is open, once Session is closed, all associated cached data is removed.

## 2) Second level cache

The second-level cache is **by default disabled,**the developer needs to enable it explicitly, and the SessionFactory object is responsible to maintain it. The second-level cache is accessible by the entire application means data hold by SessionFactory can be accessible to all the sessions. Keep in mind that, once the session factory is closed all the cache associated with that is also removed from the memory.

Let’s take an example: Suppose your application has 2 active sessions session1 and session2 respectively. Now, session1 has requested data having id=101 so that will be fetched from a database as it is the first call, and then it is stored into the second-level (SessionFactory) as well as in the first-level (session) cache also. Now, session2 requires the same data so it has also been queried with the same id=101. So this time session2 will get data from the SessionFactory, it will not going to hit the database.

# Pagination

<https://www.baeldung.com/hibernate-pagination>

# Cascade

<https://www.geeksforgeeks.org/hibernate-different-cascade-types/?ref=lbp>

# @Embeddable

* The @Embeddable annotation is used to mark a class as being embeddable, meaning its properties can be included in another class as a value type. The class marked with @Embeddable is called the embeddable class.
* The @Embeddedannotation is used to mark a field in a class as being an embeddable object, and it is used in the class that contains the embeddable object.
* By using these annotations, Hibernate can automatically persist the properties of the embeddable class within the containing class to the database table, without the need to create a separate table for the embeddable class.
* Using @Embeddable and @Embedded annotations in Hibernate allows for better data modeling, code reusability, normalization, and better performance. The annotations also allow to encapsulation of the business logic within the embeddable class

### **Benefits**

The main benefits of using the @Embeddable and @Embedded annotations in Hibernate are:

1. **Code Reusability:**You can reuse the embeddable class in multiple entities, avoiding duplication of code.
2. **Normalization:** It helps in normalizing the database by reducing the number of tables, which in turn improves the performance.
3. **Data Integrity:** It ensures data integrity by maintaining the relationship between the embeddable and the containing class.
4. **Simplicity:** It simplifies the development process by reducing the number of classes and tables required to map the data.
5. **Better Data Modelling:** It allows for better data modeling by allowing you to encapsulate the properties of an object within another object, making the data structure more intuitive and easy to understand.
6. **Ease of maintenance:**it makes the maintenance of the codebase more manageable and easy.
7. **Flexibility:** The embeddable classes can be used in multiple entities, and it also supports complex data modeling with the use of @Embedded and @AttributeOverrides.
8. **Readability:** The use of @Embeddable and @Embedded annotations makes the code more readable and self-explanatory.
9. **Performance:** it improves the performance of the application by reducing the number of joins required to fetch the data.
10. **Business logic:** It allows to encapsulation of the business logic within the embeddable class, making it more manageable and easy to understand.

# Eager/Lazy Loading

FetchType is an enumerated type in the Java Persistence API (JPA) that specifies whether the field or property should be lazily loaded or eagerly loaded. It is used in the ***javax.persistence.FetchType*** enum. In Hibernate, the FetchType is used to specify the fetching strategy to be used for an association. The FetchType can be specified for associations at the time of mapping the association. There are two FetchType options available:**LAZY and EAGER**.

***Note:****You can specify the fetch type of an association by using the fetch attribute of the****@OneToMany, @ManyToOne, @OneToOne,****or****@ManyToMany annotations.***

**LAZY:**

* This is the default FetchType in Hibernate. It means that the associated entity will be fetched only when it is accessed for the first time. This can improve performance in cases where the associated entity is not required most of the time.
* This can be more efficient than eagerly fetching the entity, especially if the entity has a lot of data and is not needed for every use of the parent entity.
* It’s important to note that using **FetchType.LAZY** can result in additional database queries being issued when the associated entity is accessed, so it may not always be the most efficient option. It’s a good idea to profile your application to determine the best fetch strategy for your use case.

**EAGER:**

* This FetchType means that the associated entity will be fetched together with the main entity when the main entity is fetched from the database. This can be useful in cases where the associated entity is always required, but can also result in a performance decrease if the associated entity is large and/or has many associations itself.
* The **FetchType.EAGER**option indicates that the associated entity should be fetched eagerly, which means that it will be fetched at the same time as the parent entity.
* Using **FetchType.EAGER** can be more efficient than using FetchType.LAZY if the associated entity is needed for most uses of the parent entity, as it avoids the need for additional database queries to fetch the associated entity when it is accessed. However, it can also be less efficient if the associated entity has a lot of data and is not needed for every use of the parent entity, as it will always be fetched along with the parent entity. It’s a good idea to profile your application to determine the best fetch strategy for your use case.

# get() and load() Method

### **1. get() method**

* **get()** method is used to retrieve a persistent object from the database. It is a member of the Session interface, and it takes the class of the object to be retrieved and the primary key of the object as arguments.
* **get()** method only hits the database if the object is not present in the session cache. If the same object is already present in the cache then it returns previously stored data from the cache.
* **get()** method returns null if there is no object present in the database.

### **2. load() method**

* **load()** method is used to retrieve an object from the database by its identifier (primary key). It is used to initialize a proxy object instead of a fully-initialized object, so it can be used to lazily load the object when it is needed.
* **load()** method does not retrieve the object from the database when it is called. Instead, it returns a proxy object that represents the object. The actual object is only retrieved from the database when it is needed, such as when a method of the object is called or a property is accessed. This technique is known as “lazy loading” and it is used to improve the performance of Hibernate by avoiding unnecessary database queries.
* **load()** method throws ObjectNotFoundException if there is no object found in the database.

# Hibernate Validator

<https://www.geeksforgeeks.org/hibernate-validator-with-example/?ref=lbp>