There are three types of inheritance mapping in hibernate   
1. Table per concrete class with unions   
2. Table per class hierarchy(Single Table Strategy)   
3. Table per subclass

@Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

1. @DiscriminatorColumn(
2. name="VEHICLE\_TYPE",
3. discriminatorType=DiscriminatorType.STRING
4. )

[view plainprint?](http://www.dineshonjava.com/p/implementing-inheritance-in-hibernate.html)

1. @DiscriminatorValue("Bike")
2. public class TwoWheeler extends Vehicle
3. {

Now adding following annotation to the FourWheeler class

[view plainprint?](http://www.dineshonjava.com/p/implementing-inheritance-in-hibernate.html)

1. @DiscriminatorValue("Car")
2. public class FourWheeler extends Vehicle
3. {

With Joined Strategy

It's highly normalized but performance is not good.  
Advantage:

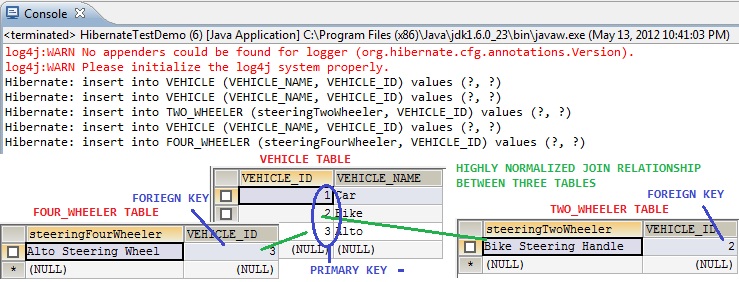
* Tables are normalized.
* Able to define NOT NULL constraint.

Disadvantage:

* Does not perform as well as SINGLE\_TABLE strategy

Using Join Strategy with the vehicle entity  
Vehicle.java

1. @Inheritance(strategy=InheritanceType.JOINED)//Highly normalized



Hibernate implements a cache region for queries resultset that integrates closely with the hibernate second-level cache.

This is an optional feature and requires additional steps in code. This is only useful for queries that are run frequently with the same parameters. First of all we need to configure below property in hibernate configuration file.

|  |  |
| --- | --- |
| 1 | <property name="hibernate.cache.use\_query\_cache">true</property> |

And in code, we need to use setCacheable(true) method of Query, quick example looks like below.

|  |  |
| --- | --- |
| 1  2  3 | Query query = session.createQuery("from Employee");  query.setCacheable(true);  query.setCacheRegion("ALL\_EMP"); |

When we use Collection API sorting algorithms to sort a collection, it’s called sorted list. For small collections, it’s not much of an overhead but for larger collections it can lead to slow performance and OutOfMemory errors. Also the entity beans should implement Comparable or Comparator interface for it to work, read more at [java object list sorting](http://www.journaldev.com/780/java-comparable-and-comparator-example-to-sort-objects).

If we are using Hibernate framework to load collection data from database, we can use it’s Criteria API to use “order by” clause to get ordered list. Below code snippet shows you how to get it.

|  |  |
| --- | --- |
| 1  2 | List<Employee> empList = session.createCriteria(Employee.class)  .addOrder(Order.desc("id")).list(); |

Ordered list is better than sorted list because the actual sorting is done at database level, that is fast and doesn’t cause memory issues.

Hibernate uses [Reflection API](http://www.journaldev.com/1789/java-reflection-tutorial-for-classes-methods-fields-constructors-annotations-and-much-more) 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 getHibernateException.

@NamedQuery(name = "UserDetails.byId", query = "from UserDetails where UserId = ?")

Hibernate Named Query helps us in grouping queries at a central location rather than letting them scattered all over the code.  
Hibernate Named Query syntax is checked when the hibernate session factory is created, thus making the application fail fast in case of any error in the named queries.  
Hibernate Named Query is global, means once defined it can be used throughout the application.

However one of the major disadvantage of Named query is that it’s hard to debug, because we need to find out the location where it’s defined.

List cats = sess.createCriteria(Cat.class)

.add( Restrictions.like("name", "Fritz%") )

.add( Restrictions.between("weight", minWeight, maxWeight) )

.list();

List cats = sess.createCriteria(Cat.class)

.add( Restrictions.like("name", "F%")

.addOrder( Order.asc("name") )

.addOrder( Order.desc("age") )

.setMaxResults(50)

.list();

List results = session.createCriteria(Cat.class)

.setProjection( Projections.projectionList()

.add( Projections.rowCount() )

.add( Projections.avg("weight") )

.add( Projections.max("weight") )

.add( Projections.groupProperty("color") )

)

.list();

Hibernate Native SQL Example

We can use Session.createSQLQuery(String query) to create the SQLQuery object and execute it.