**Difference between Update and Merge methods In Hibernate**

Often times, you will notice Hibernate developers mix use of [session.update()](http://www.simplecodestuffs.com/hibernate-update-query-example/" \t "_blank) and session.merge()

At first look both update() and merge() methods seems similar because both of them are used to convert the object which is in detached state into persistence state, but the major difference between update and merge is that update method cannot be used when the same object exists in the session. Let’s look at those difference with simple example.  
   
Example :-

1. **Student** current = (**Student**)session.**get**(**Student**.**class**, 100);
2. **System**.**out**.println("Before merge: " + current.getName());
3. **Student** changed = **new** **Student**();
4. changed.setId(100);
5. changed.setName("Changed new Name");
6. // session.update(changed); // Throws NonUniqueObjectException
7. session.merge(changed);
8. **System**.**out**.println("After merge: " + current.getName());

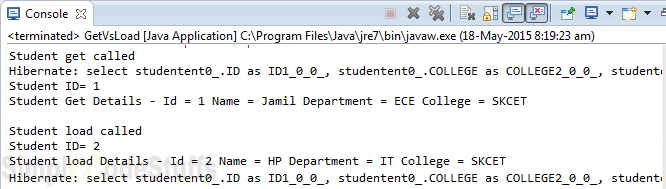
Explanation

In the above program I have loaded a Student object of ID ‘100’ at line no 1. After that I have created a new Student object ‘changed’ with same ID ‘100’. Now if I try to call update method on this ‘changed’ object, then Hibernate will through a NonUniqueObjectException, because the same object (Student) with Id ‘100’ already exists in session.  
   
Now at line no ‘7’, I have called session.merge(changed); this merge() will work fine and the name will get changed and saved into the database. After this on printing the current object, it will print the latest changed value, since when the merge occurs the value loaded in session gets changed.  
   
Done and done! So there you have it, you now know the exact difference between merge() and update() method in Hibernate framework and actually the usage of merge() methods will come into picture when ever we try to load the same object again and again into the database.

**Different Between Session.Get() and Session.Load() in Hibernate**

Often times, you will notice Hibernate developers mix use of **session.get()** and **session load()**  
   
At first look both get() and load() seems similar because both of them fetch the data from database, however there are few differences between them, let’s look at those difference with simple example.

**package** util;  
  
**import** entity.**StudentEntity**;  
  
**import** org.hibernate.**Session**;  
**import** org.hibernate.**SessionFactory**;  
**import** org.hibernate.boot.registry.**StandardServiceRegistryBuilder**;  
**import** org.hibernate.cfg.**Configuration**;  
**import** org.hibernate.service.**ServiceRegistry**;  
  
**public** **class** **GetVsLoad** {  
  
        **public** **static** **void** main(**String**[] args) {  
  
                **Configuration** cf = **new** **Configuration**().configure("hibernate.cfg.xml");  
  
                **StandardServiceRegistryBuilder** srb = **new** **StandardServiceRegistryBuilder**();  
                srb.applySettings(cf.getProperties());  
                **ServiceRegistry** sr = srb.build();  
                **SessionFactory** sf = cf.buildSessionFactory(sr);  
  
                **Session** session = sf.openSession();  
          
                // Get Example  
                **System**.out.println("Student get called");  
  
                **StudentEntity** student1 = (**StudentEntity**) session.get(**StudentEntity**.**class**, 1);  
                **System**.out.println("Student ID= " + student1.getId());  
                **System**.out.println("Student Get Details - " + student1 + "n");  
  
                // load Example  
                **System**.out.println("Student load called");  
                **StudentEntity** student2 = (**StudentEntity**) session.load(**StudentEntity**.**class**, 2);  
                **System**.out.println("Student ID= " + student2.getId());  
                **System**.out.println("Student load Details - " + student2 + "n");  
  
                // Close resources  
                session.close();  
                sf.close();  
        }  
}

  
   
From the output it’s clear that get() returns the object by fetching it from database whereas load() method will not hit the database (no select statement in output before printing ID) to retrieve the StudentEntity object, it will return a StudentEntity proxy object – a fake object created by hibernate with given identifier value(ID = 2).  
   
In the above example we have called StudentEntity student2 = (StudentEntity) session.load(StudentEntity.class, 2); So here hibernate will create one fake StudentEntity object in the memory with id 2, but the other properties of StudentEntity class will not be initialized.  
   
This method loads the data from database only when we try to access other properties of the StudentEntity object(Ex:- name, department, college). In the above example the DB hit takes place only after we try to call other member variable of StudentEntity object.

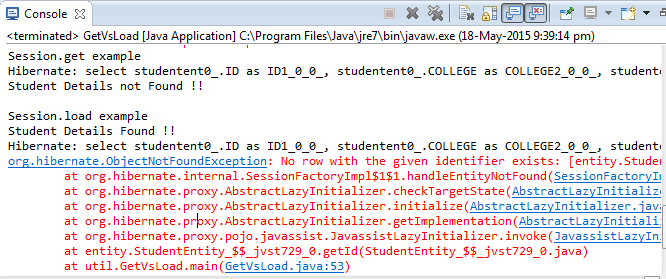
#### Now let’s try to fetch data that doesn’t exists in the database

**Get Example**

// Get Example  
**try** {  
        **System**.out.println("Session.get example");  
        **StudentEntity** std = (**StudentEntity**) session.get(**StudentEntity**.**class**, 100);  
  
        **if** (std == **null**) {  
        **System**.out.println("Student Details not Found !! ");      
        }   
        **else**  
        {  
                **System**.out.println("Student Details Found !! ");          
                **System**.out.println("Student GET ID= " + std.getId());  
                **System**.out.println("Student Get Details - " + std + "n");  
        }  
} **catch** (**Exception** e) {  
        e.printStackTrace();  
}

**Load Example**

// load Example  
**try** {  
        **System**.out.println("nSession.load example");  
        **StudentEntity** std2 = (**StudentEntity**) session.load(**StudentEntity**.**class**, 100);  
  
        **if** (std2 == **null**) {  
                **System**.out.println("Student Details not Found !! ");  
        }  
        **else**  
        {  
                **System**.out.println("Student Details Found !! ");  
                **System**.out.println("Student LOAD ID= " + std2.getId());  
                **System**.out.println("Student load Details - " + std2 + "n");  
        }  
          
} **catch** (**Exception** e) {  
        e.printStackTrace();  
}

Above code produces following output.  
   
  
   
In the above program when we use get() to retrieve data that doesn’t exists, it returns null. That makes sense because it try to load the data as soon as it’s called.  
   
But With load(), we are able to print the ID; but as soon as we try to access other fields, it fires database query and throws org.hibernate.ObjectNotFoundException when there is no record found with the given identifier.  
   
**Another must read**  
[Hibernate Eager vs Lazy Fetch Type](http://www.simplecodestuffs.com/hibernate-eager-vs-lazy-fetch-type/)

#### Special cases

If get(StudentEntity.class, 1) is called first and later again load(StudentEntity.class, 1) is called and if the record with primary key 1 exists, the s1 in the both the cases contains a real object. Why? We know get() method returns a StudentEntity object and load() returns s proxy object. With get(), a real object exists in the cache memory, the same object is returned to load() also.  
If the reverse happens? If load() is called first and then get()? In both cases, s1 contains a proxy object because first load() returns a proxy and remains in the cache memory.

#### Summary

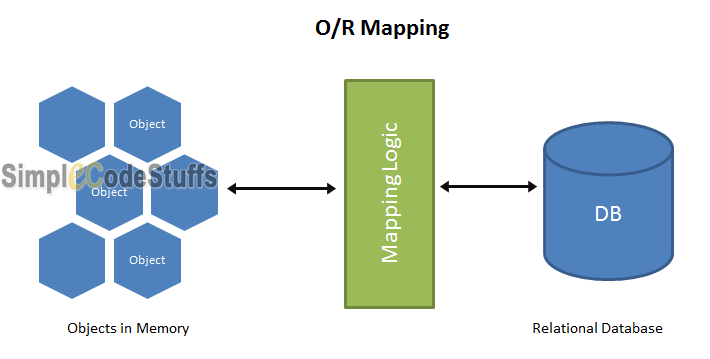
* get() loads the data from database as soon as it’s called whereas load() returns a proxy object and loads data only from database when it’s actually required, so load() is better because it support lazy loading.
* Since load() throws exception when data is not found, we should use it only when we know data exists.
* We should not use load method to determine if an instance exists (use get() instead). Use this only to retrieve an instance that you assume exists, where non-existence would be an actual error.
* Both methods are used to retrieve only one Java object (record) from the database (for multiple records, we have, list() and iterate() methods).

# **Introduction to Hibernate**

In this to we will learn about what hibernate is, and how to use hibernate in our application and its features.

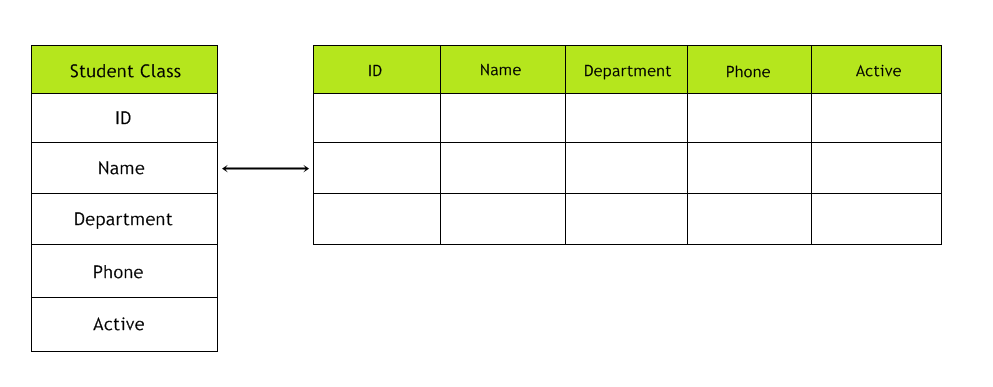
#### What is hibernate ?

* It called as ORM tool (we will learn in subsequent tut)
* Used in the data layer to save date into the database
* Implements JPA (JPA is a set of standards which is prescribed for doing any operation in database.



#### 1. Problem which Hibernate solves

Mapping member variables to column

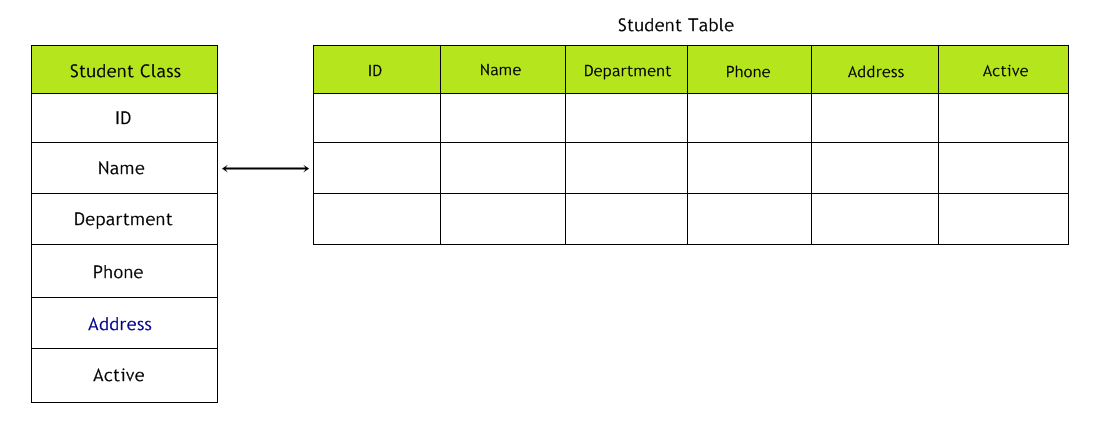
Let take a Student class as an example, which has the following fields ID, Name, Department, Phone no, Address, Active.[](http://i0.wp.com/www.simplecodestuffs.com/wp-content/uploads/2014/04/Hibrenet-introduction.png)In a running application we may have lot of such objects, now to save the data in db; I need to create a Student table, in which I will save the Student objects as rows. So here N no of objects equal to N no of row in the table.

This implies that the class corresponds to the table and object of the class corresponds to the rows in the table. In java application, we will have a JDBC to connect to the db, and we will take this Student object, create an insert sql query and do an insert, Now all these data will be stored in db.

Now to select these data, we will create a select query to pull to the records and to store the records we will create a Student object to. So in Java we have data in form of objects, but there is no object in table, So java entity needs to be converted into a table entity, (i.e.) records.

The way we normally convert is by taking each of the values and map to its rows, this mapping is more tedious as we need to convert each object into SQL quires for saving, and for retrieving I have to convert a result set into corresponding java objects.

2. Mapping relationship

For example Student table has a column which is the primary key of another table (Address).  
[](http://i1.wp.com/www.simplecodestuffs.com/wp-content/uploads/2014/04/Hibernet-example.png)  
Here in java side care should be taken to map Student object to Student table and Address object to Address table.

3. Handling data types

Example:

To check whether the Student is active or not, in java side, we will create a Boolean variable which hold the value true or false, here in case of Database, we cannot have a column with type Boolean, for this scenario we can either have a char type or int type. So now we have to handle this data type conversion by ourselves.

4. Managing the changes to object state.

Suppose if we want to update any of the field n DB, then we have to pull the object from db, store it in Student object then write a java code to trigger update query.

The above 4 are the most common problem which we face in most of the java application. In our next article we shall learn to set up Hibernate in eclipse.

# **Hibernate advantages and disadvantages**

#### Advantages of using Hibernate

* Traditionally Programmers need to write functions that convert the object to query and result set to object. To overcome these difficulties, Hibernate provides a mechanism to maps Java classes to database tables and from Java data types to SQL data types and relieve the developer from 95% of common data persistence related programming tasks. [Read more..](http://www.simplecodestuffs.com/introduction-to-hibernate/)
* Database independent – Hibernate has its own query language, i.e **Hibernate Query Language(HQL)** which is independent of database. HQL is an object-oriented extension to SQL, So we no need to write code specific to database.
* No knowledge of SQL is needed. For example **saveOrUpdate**(entityObject) takes care of insertion in case of Hibernate.
* Takes care of dependencies between tables and does **join queries**.
* Query tuning is not required in Hibernate as it is automatically achieved by using **Criteria** queries
* Hibernate supports **first level cache** by default. This reduces the number of round trips between application and database and hence increases performance.
* [Hibernate supports Inheritance](http://www.simplecodestuffs.com/inheritance-mapping-in-hibernate-introduction/) – In hibernate if we save the derived class object, then its base class object will also be stored into the database.
* Hibernate supports relationships like **One-To-Many,One-To-One, Many-To-Many-to-Many, Many-To-One**
* Hibernate supports **Collections** like **List,Set,Map**
* Hibernate has capability to generate **primary keys** automatically while we are storing the records into database.
* Maintains transactions commit and rollback.
* Maintains database connection pool.
* Hibernate does not require an application server to operate.

#### Supported Databases:

Hibernate supports almost all the major RDBMS. Following is list of few of the database engines supported by Hibernate.

* HSQL Database Engine
* DB2/NT
* MySQL
* PostgreSQL
* FrontBase
* Oracle
* Microsoft SQL Server Database
* Sybase SQL Server
* Informix Dynamic Server

#### Disadvantages of hibernate

* If it is a small project with few tables(< 10 tables) , then there is no need for a ORM framework. In such case a normal JDBC is enough.
* Hibernate has a performance cost as it adds a layer over jdbc. Also with Hibernate lot of table configuration information (**hbm or annotations**) is read at start up adding to the start up time of your application.
* **Not suitable for Batch processing**– It advisable to use pure JDBC for batch processing.

# **steps to be followed to use Hibernate in Java**

In our previous tutorial we have learned about [Simple Hibernate Application Requirements](http://www.simplecodestuffs.com/generic-hibernate-application-requirements/). In this article we shall explore about the exact flow to use hibernate in a java application.  
   
Following are the Steps to be followed to Use Hibernate in Java

#### 1. Import Hibernate API

Import the following hibernate API.

import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
import org.hibernate.Transaction;  
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;  
import org.hibernate.cfg.Configuration;  
import org.hibernate.service.ServiceRegistry;

#### 2. Loading Configuration files

To load hibernate configuration xml, we need to create object of Configuration class and we need to call configure() method in that class.  
   
**Example:**

**Configuration** cf = **new** **Configuration**().configure("hibernate.cfg.xml");

Once configure method is called, the following actions takes place:

* The configuration object cf will read configuration details from hibernate.cfg.xml file and Stores the data in different variables of a high level hibernate object called SessionFactory.
* Now this SessionFactory object sf contains all the data regarding the configuration file

So in java we get this configuration details by using the following code.

**StandardServiceRegistryBuilder** srb = **new** **StandardServiceRegistryBuilder**();  
srb.applySettings(cf.getProperties());  
**ServiceRegistry** sr = srb.build();  
**SessionFactory** sf = cf.buildSessionFactory(sr);

**Note:** Once we load the configuration file, the mapping file will be loaded automatically as we have configured the mapping xml in the hibernate configuration file.

#### 3. Creating an object of session

To open a database connection in hibernate, we need to call openSession() method of SessionFactory, now assign this connection to Session interface.

**Session** session = sf.openSession();  
//sf = Object of SessfionFactory

#### 4. Create a logical transaction

Hibernate needs a logical Transaction to perform insert, update and delete operations into any database. In order to begin a logical transaction in hibernate we need to call beginTransaction() method of Session Interface and assign it to Transaction interface.

**Transaction** tx = session.beginTransaction();  
//session = Object of Session Interface

**Note:** For selecting an object from the database we do not require any logical transaction in hibernate.

#### 5. Use Session Interface methods

We can use the methods given by Session Interface, to move the objects from java to database and from database to java  
   
[session.save(s)](http://www.simplecodestuffs.com/hibernate-4-hello-world-example-in-eclipse-using-annotation/) – Insert object ‘s’ into database  
[session.delete(s)](http://www.simplecodestuffs.com/hibernate-delete-query-example/) – Delete object ‘s’ from database  
[session.update(s)](http://www.simplecodestuffs.com/hibernate-update-query-example/) – Update object ‘s’ in the database  
[session.load(s)](http://www.simplecodestuffs.com/example-on-hibernate-select-query/) – Select object ‘s’ from database

#### 6. Commit Transaction

Now we need to call commit() method of Transaction to perform commit operation using tx.commit();

#### 7. Close Session and SessionFactory

As discussed earlier, when we open a session, a connection to database is created, so once the transaction is complete, we must close that connection via session.close() code and finally we have to close the SessionFactory via sf.close()code.  
   
So the final flow of any hibernate application will be as below

Configuration  
SessionFactory  
Open Session  
Begin Transaction  
Session  
Commit Transaction  
Close Session and SessionFactory

# [Hibernate Select Query](http://www.simplecodestuffs.com/example-on-hibernate-select-query/)

In previous article we learnt to implement [Hibernate 4 Insert Query example in Eclipse](http://www.simplecodestuffs.com/hibernate-4-hello-world-example-in-eclipse-using-annotation/), In this article I will implement an example on loading the object from the database(Select Query) using Hibernate.

#### Entity class

**package** entity;  
  
**import** javax.persistence.**Column**;  
**import** javax.persistence.**Entity**;  
**import** javax.persistence.**GeneratedValue**;  
**import** javax.persistence.**Id**;  
**import** javax.persistence.**Table**;  
  
@Entity  
@Table(name = "STUDENT")  
**public** **class** **StudentEntity** {  
  
        @Id  
        @Column(name = "ID")  
        **private** **int** id;  
          
        @Column(name = "NAME")  
        **private** **String** name;  
          
        @Column(name = "DEPARTMENT")  
        **private** **String** department;  
          
        @Column(name = "COLLEGE")  
        **private** **String** college;  
  
// Create Getters and Setters  
}

**Note:** I have explained about every annotations used in the above file in the article [Generic Hibernate Application Requirements](http://www.simplecodestuffs.com/generic-hibernate-application-requirements/)

#### Hibernate Configuration file

Create a new XML file and give this new configuration file the default name hibernate.cfg.xml and place it src directory of your project.  
   
File: hibernate.cfg.xml

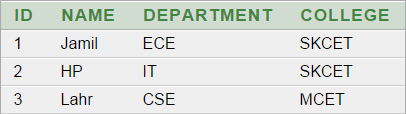
<!DOCTYPE hibernate-configuration PUBLIC  
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  
  
<hibernate-configuration>  
  
<session-factory>  
        <!-- Database connection settings -->  
        <property name="hibernate.connection.driver\_class">  
                  oracle.jdbc.driver.OracleDriver  
        </property>  
        <property name="hibernate.connection.username">system</property>  
        <property name="hibernate.connection.password">admin</property>  
        <property name="hibernate.connection.url">  
                   jdbc:oracle:thin:@xxx.x.x.x:1521:XE  
        </property>  
  
        <!-- SQL dialect -->  
        <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>  
  
        <!-- Echo all executed SQL to sysout -->  
        <property name="show\_sql">true</property>  
  
        <!-- Drop and re-create the database schema on startup -->  
        <property name="hibernate.hbm2ddl.auto">update</property>  
        <!-- Map Entity Class -->  
    <mapping class="entity.StudentEntity"></mapping>  
  
</session-factory>  
</hibernate-configuration>

#### Hibernate Utility

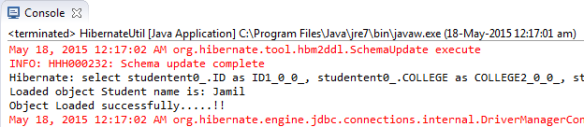
1. **package** util;
2. **import** org.hibernate.**Session**;
3. **import** org.hibernate.**SessionFactory**;
4. **import** org.hibernate.boot.registry.**StandardServiceRegistryBuilder**;
5. **import** org.hibernate.cfg.**Configuration**;
6. **import** org.hibernate.service.**ServiceRegistry**;
7. **import** entity.**StudentEntity**;
8. **public** **class** **HibernateUtil** {
9. **public** **static** **void** main(**String**[] args) {
10. **Configuration** cf = **new** **Configuration**().configure("hibernate.cfg.xml");
12. **StandardServiceRegistryBuilder** srb = **new** **StandardServiceRegistryBuilder**();
13. srb.applySettings(cf.getProperties());
14. **ServiceRegistry** sr = srb.build();
15. **SessionFactory** sf = cf.buildSessionFactory(sr);
16. **Session** session = sf.openSession();
17. **StudentEntity** std = (**StudentEntity**) session.load(**StudentEntity**.**class**, **new** **Integer**(1));
18. // For loading Transaction object is not necessary
19. **System**.**out**.println("Loaded object Student name is: " + std.getName());
20. **System**.**out**.println("Object Loaded successfully.....!!");
21. session.close();
22. sf.close();
23. }
24. }

In the article [Steps to be followed to use Hibernate in Java](http://www.simplecodestuffs.com/steps-to-be-followed-to-use-hibernate-in-java/), I have explained the configuration/code used in above program in detail.

#### Data in Database



#### Run it – Eclipse Console



#### Note

In hibernate we have 2 methods to load the object from the database, they are get and load. To know the difference between get and load method read the article [here](http://www.simplecodestuffs.com/different-session-get-session-load-hibernate/).

# **Hibernate Delete Query example**

In previous article we learnt to implement [Hibernate Select Query](http://www.simplecodestuffs.com/example-on-hibernate-select-query/) example in Eclipse, In this article I will implement an example on delete the object from the database(Delete Query) using Hibernate.

#### Entity class

**package** entity;  
  
**import** javax.persistence.**Column**;  
**import** javax.persistence.**Entity**;  
**import** javax.persistence.**GeneratedValue**;  
**import** javax.persistence.**Id**;  
**import** javax.persistence.**Table**;  
  
@Entity  
@Table(name = "STUDENT")  
**public** **class** **StudentEntity** {  
  
        @Id  
        @Column(name = "ID")  
        **private** **int** id;  
          
        @Column(name = "NAME")  
        **private** **String** name;  
          
        @Column(name = "DEPARTMENT")  
        **private** **String** department;  
          
        @Column(name = "COLLEGE")  
        **private** **String** college;  
  
// Create Getters and Setters  
}

**Note:** I have explained about every annotations used in the above file in the article [Generic Hibernate Application Requirements](http://www.simplecodestuffs.com/generic-hibernate-application-requirements/)

#### Hibernate Configuration file

Create a new XML file and give this new configuration file the default name hibernate.cfg.xml and place it src directory of your project.  
   
File: hibernate.cfg.xml

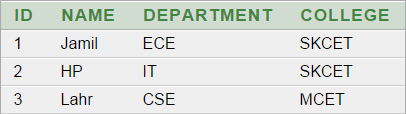
<!DOCTYPE hibernate-configuration PUBLIC  
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  
  
<hibernate-configuration>  
  
<session-factory>  
        <!-- Database connection settings -->  
        <property name="hibernate.connection.driver\_class">  
                  oracle.jdbc.driver.OracleDriver  
        </property>  
        <property name="hibernate.connection.username">system</property>  
        <property name="hibernate.connection.password">admin</property>  
        <property name="hibernate.connection.url">  
                   jdbc:oracle:thin:@xxx.x.x.x:1521:XE  
        </property>  
  
        <!-- SQL dialect -->  
        <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>  
  
        <!-- Echo all executed SQL to sysout -->  
        <property name="show\_sql">true</property>  
  
        <!-- Drop and re-create the database schema on startup -->  
        <property name="hibernate.hbm2ddl.auto">update</property>  
        <!-- Map Entity Class -->  
    <mapping class="entity.StudentEntity"></mapping>  
  
</session-factory>  
</hibernate-configuration>

#### Hibernate Utility

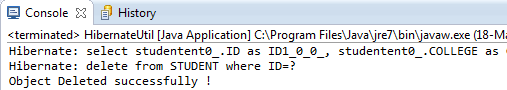
1. **package** util;
2. **import** org.hibernate.**Session**;
3. **import** org.hibernate.**SessionFactory**;
4. **import** org.hibernate.**Transaction**;
5. **import** org.hibernate.boot.registry.**StandardServiceRegistryBuilder**;
6. **import** org.hibernate.cfg.**Configuration**;
7. **import** org.hibernate.service.**ServiceRegistry**;
8. **import** entity.**StudentEntity**;
9. **public** **class** **HibernateUtil** {
10. **public** **static** **void** main(**String**[] args) {
11. **Configuration** cf = **new** **Configuration**().configure("hibernate.cfg.xml");
12. **StandardServiceRegistryBuilder** srb = **new** **StandardServiceRegistryBuilder**();
13. srb.applySettings(cf.getProperties());
14. **ServiceRegistry** sr = srb.build();
15. **SessionFactory** sf = cf.buildSessionFactory(sr);
16. **Session** session = sf.openSession();
17. **StudentEntity** std = (**StudentEntity**) session.load(**StudentEntity**.**class**, **new** **Integer**(2));
18. **Transaction** tx = session.beginTransaction();
19. session.**delete**(std);
20. tx.commit();
21. **System**.**out**.println("Object Deleted successfully !");
22. session.close();
23. sf.close();
24. }
25. }

In the above program we are deleting an object, which is already persisted (inserted) in the database; So in order to delete an persisted object, we need to load that object from the database and pass that object to the delete() method of Sessioninterface, now hibernate delete that object whenever the transaction is committed.  
   
In the article [Steps to be followed to use Hibernate in Java](http://www.simplecodestuffs.com/steps-to-be-followed-to-use-hibernate-in-java/), I have explained the configuration/code used in above program in detail.

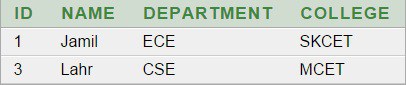
#### Data in DB before running the program



#### Run it – Eclipse Console



#### Data in DB after running the program



# **Hibernate update query example**

In previous article we learnt to implement [Hibernate delete Query example](http://www.simplecodestuffs.com/hibernate-delete-query-example/) in Eclipse, in this article I will implement a program to update an object which is already persisted in the database.

#### Entity class

**package** entity;  
  
**import** javax.persistence.**Column**;  
**import** javax.persistence.**Entity**;  
**import** javax.persistence.**GeneratedValue**;  
**import** javax.persistence.**Id**;  
**import** javax.persistence.**Table**;  
  
@Entity  
@Table(name = "STUDENT")  
**public** **class** **StudentEntity** {  
  
        @Id  
        @Column(name = "ID")  
        **private** **int** id;  
          
        @Column(name = "NAME")  
        **private** **String** name;  
          
        @Column(name = "DEPARTMENT")  
        **private** **String** department;  
          
        @Column(name = "COLLEGE")  
        **private** **String** college;  
  
// Create Getters and Setters  
}

**Note:** I have explained about every annotations used in the above file in the article [Generic Hibernate Application Requirements](http://www.simplecodestuffs.com/generic-hibernate-application-requirements/)

#### Hibernate Configuration file

File: hibernate.cfg.xml

<hibernate-configuration>  
  
<session-factory>  
   <!-- Database connection settings -->  
   <property name="hibernate.connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>  
   <property name="hibernate.connection.username">system</property>  
   <property name="hibernate.connection.password">admin</property>  
   <property name="hibernate.connection.url">jdbc:oracle:thin:@127.0.0.1:1521:XE</property>  
  
   <!-- SQL dialect -->  
   <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>  
  
   <!-- Echo all executed SQL to sysout -->  
   <property name="show\_sql">true</property>  
  
   <!-- Create/Update the database schema on startup -->  
   <property name="hibernate.hbm2ddl.auto">update</property>  
   <!-- Mapping file -->  
   <mapping resource="Student.hbm.xml" />  
  
</session-factory>  
  
</hibernate-configuration>

#### Hibernate Utility

1. **package** util;
2. **import** org.hibernate.**Session**;
3. **import** org.hibernate.**SessionFactory**;
4. **import** org.hibernate.**Transaction**;
5. **import** org.hibernate.boot.registry.**StandardServiceRegistryBuilder**;
6. **import** org.hibernate.cfg.**Configuration**;
7. **import** org.hibernate.service.**ServiceRegistry**;
8. **import** model.**Student**;
9. **public** **class** **HibernateUtil** {
10. **public** **static** **void** main(**String**[] args) {
11. **Configuration** cf = **new** **Configuration**().configure("hibernate.cfg.xml");
12. **StandardServiceRegistryBuilder** srb = **new** **StandardServiceRegistryBuilder**();
13. srb.applySettings(cf.getProperties());
14. **ServiceRegistry** sr = srb.build();
15. **SessionFactory** sf = cf.buildSessionFactory(sr);
16. **Session** session = sf.openSession();
17. **Student** std = (**Student**) session.load(**Student**.**class**, **new** **Integer**(2));
18. **Transaction** tx = session.beginTransaction();
19. // std.setId(3); // We Should not update "id"
20. std.setName("Ameer");
21. std.setCollege("PSG");
22. tx.commit(); // Update method will be called implicitly.
23. **System**.**out**.println("Object Updated successfully !");
24. session.close();
25. sf.close();
26. }
27. }

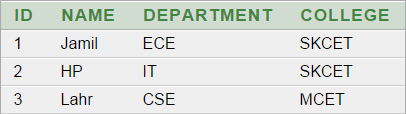
#### Hibernate Cache

* Whenever an object is loaded from database, hibernate stores the loaded object in cache memory which works in session scope.
* Now if we do any modifications to the loaded object, then these modification are stored only in the object maintained by cache memory. Even if we modify the loaded object for multiple times then also the modifications will be stored only in the cached object.
* But once we call transactions commit() method then hibernate will check whether there are any changes between the object present in the database and the object stored in the cache, now the if changes exists then hibernate will automatically call its update method internally and updates the student object in database.

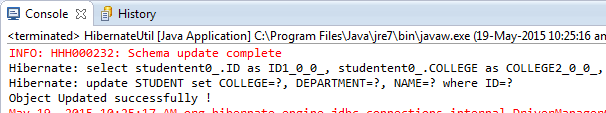
#### Program Explanation

* Line no 28: We should not update id, because we have loaded the object from the database only using this id member variable (See line no 25). So if we update this field, then hibernate will throws NonUniqueObjectException.
* At line no 29, I have updated name property of student object.
* Line no 31: So once commit() method is triggered from program, then hibernate will automatically call update method internally and updates the student object.

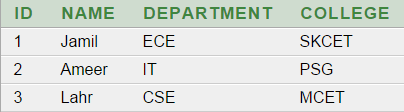
#### Data in DB before running the program



#### Run it – Eclipse Console



#### Data in DB after running the program



# **Hibernate Life Cycle**

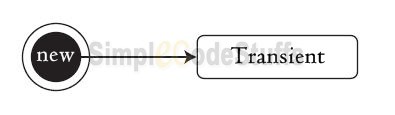
Hibernate is picky about your Java objects. Hibernate prefers your objects to be in a certain “state”, and there are four different states that exist inside of the hibernate life cycle. They are-

1. Transient
2. Persistent
3. Detached
4. Removed

Once you have a firm grasp of the different states that an object can be in Hibernate, you’ll be well on your way to mastering the Hibernate framework. Now let’s get this Hibernate life cycle lesson started.

#### Transient

When ever an object of a pojo class is Created(instantiated) using the new operator then it will be in the Transient state; this object is not associated with any Hibernate Session.



This object don’t have any association with any database table row. In other words any modification in data of transient state object doesn’t have any impact on the database table.  

#### Persistent

When the object is in persistent state, then it represent one row of the database, and it is associated with the unique Session. Hibernate will detect any changes made to an object in persistent state and synchronize the state with the database when the unit of work completes.    
   
You can create persistent objects via two ways:

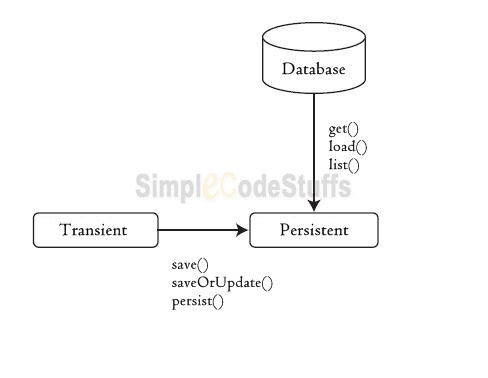
1. Loading the object from the database via Hibernate APIs
2. Saving the object to the database via Hibernate APIs

**Ways to Save an Object**  
   
Hibernate has a few different ways to save an object to the database, but the two main ways are as follows:

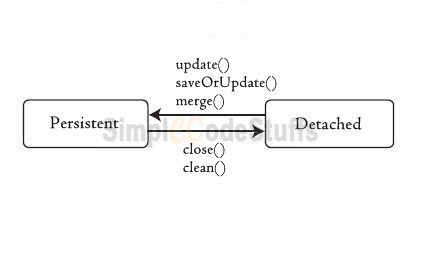
1. [save()](http://www.simplecodestuffs.com/hibernate-4-hello-world-example-in-eclipse/)
2. saveOrUpdate()
3. persist();

Invoking either of these Hibernate methods will shift your transient object into the persistent state (so long as the save is successful).  
   
**Ways to Load an Object**  
   
There are quite a few ways to load an object from a database. Here’s a couple that I use the most:

1. get()
2. [load()](http://www.simplecodestuffs.com/example-on-hibernate-select-query/)
3. list()



#### Detached

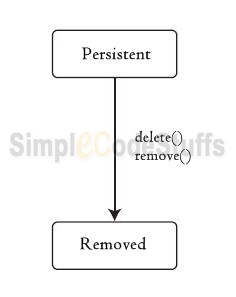
if we want to move an object from persistent to detached state, we need to close the session or clear the cache of the session.  


* Here the reference to the object is still valid and the detached instance can be modified in this state.
* A detached instance can be reattached to a new Session at a later point in time, making it (and all the modification) persistent again.

This reattaching of object from detached  to persistent state can be done by calling the following methods –

* [update()](http://www.simplecodestuffs.com/hibernate-update-query-example/)
* merge()
* saveOrUpdate()
* lock() – It is reattached but not saved.

#### Removed

A persistent object is considered to be in the removed state when a [delete()](http://www.simplecodestuffs.com/hibernate-delete-query-example/) or remove() operation is called on it. Note that Once you’ve deleted an object and moved to the “removed” state, you should no longer use that particular object for any reason.  


# **Auto Generate Primary key in Hibernate**

In our [early tutorials](http://www.simplecodestuffs.com/hibernate-4-hello-world-example-in-eclipse-using-annotation/) I have used @Id annotation to mark a particular field as primary key, this Primary key is used to fetch data via [get or load](http://www.simplecodestuffs.com/different-session-get-session-load-hibernate/) method. And we set the value of primary key manually each time when we are about to create a new record.  
   
In this article we shall learn to Auto Generate this Primary key value in Hibernate. Before getting into the above topic, let us have a discussion about Natural key and Surrogate key.

#### Natural Key

A natural key is a single column or set of columns which are made up of real data. When I say “real data” I mean data that has meaning and occurs naturally in the world of data. Some examples of natural keys values are Email id, Social Security Number etc.

#### Surrogate key

Surrogate key is “a system-generated value used to uniquely identify a row”. Its value is typically generated at run time right before the record is inserted into a table. Unlike natural keys, Surrogate key have no business meaning. Surrogate keys are commonly a numeric number.  
   
So if we are having a natural key, it would make sense for us to provide the value via application, because it’s of business significance and we need to control its value.  
   
But if it’s a Surrogate key then it’s not necessary that we have to provide it via our application. We can just ask hibernate to do the job for us, because it does not matter for us that what its value is going to be, we just need to make sure that its value had to be unique and not null.  
   
To indicate hibernate to generate value for a primary key (Surrogate key) while inserting a new record in a table, we have to use @GeneratedValue. This annotation makes the column to be auto generated.  
   
Example:

#### Entity Class

1. **package** entity;
2. **import** javax.persistence.**Column**;
3. **import** javax.persistence.**Entity**;
4. **import** javax.persistence.**GeneratedValue**;
5. **import** javax.persistence.**Id**;
6. **import** javax.persistence.**Table**;
7. @Entity
8. @Table(name = "STUDENT")
9. **public** **class** **StudentEntity** {
10. @Id
11. @GeneratedValue
12. @Column(name = "ID")
13. **private** **int** id;
14. @Column(name = "NAME")
15. **private** **String** name;
16. @Column(name = "DEPARTMENT")
17. **private** **String** department;
18. @Column(name = "COLLEGE")
19. **private** **String** college;
21. //Create Getters and Setters
22. }

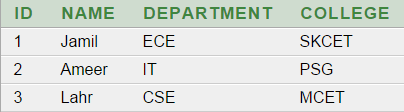
#### Hibernate Utility

Create the Main class to run the example.

**public** **class** **HibernateUtil** {  
  
        **public** **static** **void** main(**String**[] args) {  
  
        **Configuration** cf = **new** **Configuration**().configure("hibernate.cfg.xml");  
  
        **StandardServiceRegistryBuilder** srb = **new** **StandardServiceRegistryBuilder**();  
        srb.applySettings(cf.getProperties());  
        **ServiceRegistry** sr = srb.build();  
        **SessionFactory** sf = cf.buildSessionFactory(sr);  
  
        **Session** session = sf.openSession();  
        **Transaction** tx = session.beginTransaction();  
          
        **StudentEntity** student1 = **new** **StudentEntity**();   
  
        student1.setName("Jamil");  
        student1.setDepartment("ECE");  
        student1.setCollege("SKCET");  
          
        **StudentEntity** student2 = **new** **StudentEntity**();   
  
        student2.setName("Hp");  
        student2.setDepartment("IT");  
        student2.setCollege("SKCET");  
  
        **StudentEntity** student3 = **new** **StudentEntity**();   
  
        student3.setName("Lahr");  
        student3.setDepartment("CSE");  
        student3.setCollege("MCET");  
  
        session.save(student1);   
        session.save(student2);  
        session.save(student3);  
  
        tx.commit();   
        session.close();  
        sf.close();  
        }  
}

File hibernate.cfg.xml is same as in [HelloWorld tutorial](http://www.simplecodestuffs.com/hibernate-4-hello-world-example-in-eclipse-using-annotation/" \t "_blank).

#### Database output

On running the above program, we get the following values in database.  
  
   
Here the values in column ID is auto generated via @GeneratedValue.  
   
In our next article we shall learn about configuring [Composite Primary Keys](http://www.simplecodestuffs.com/composite-primary-keys-in-hibernate/) In Hibernate.

# **Composite Primary Keys In Hibernate**   If the table has a primary key then in Entity class we configure that column using [@Id annotation](http://www.simplecodestuffs.com/auto-generate-primary-key-in-hibernate/). Even when the table doesn’t need a primary key, we must configure one column as id (one primary key is must).   Now 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 , then in order to configure these primary key columns we need to create a new @Embeddable class containing the PK fields:

@Embeddable  
**public** **class** **RegistrationId** **implements** **Serializable**{  
  
        @Column(name = "STUDENT\_ID")  
        **private** **int** studentId;  
  
        @Column(name = "DEPARTMENT")  
        **private** **String** department;  
  
        // Create getters and Setters   
}

And we should use it in the @Entity as a @EmbeddedId:

@Entity  
@Table(name = "STUDENT")  
**public** **class** **StudentEntity** {  
  
        @EmbeddedId  
        **private** **RegistrationId** regid;  
        ...  
        // Create Getters and Setters  
}

To persist the entity:

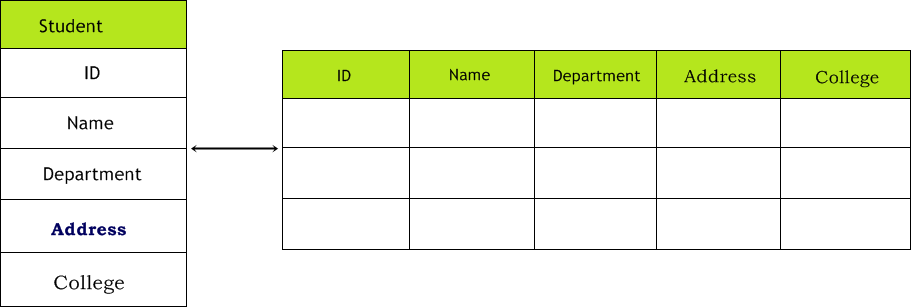
**RegistrationId** regId = **new** **RegistrationId**();  
   regId.setStudentId(1);  
   regId.setDepartment("ECE");  
  
   **StudentEntity** student = **new** **StudentEntity**();   
   student.setRegid(regId);  
          
   session.save(student);

The following rules must apply for composite primary keys:

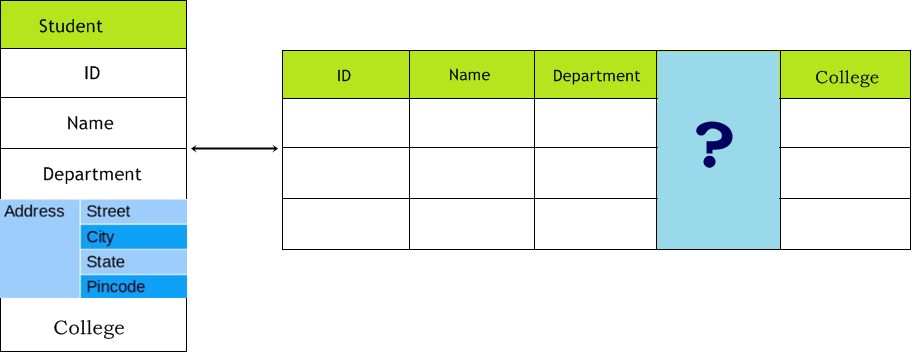
* The primary key class must be public.
* If property-based access is used, the properties of the primary key class must be public or protected.
* The primary key class must be serializable. We shall explore the reason for implementing this interface in our upcoming tutorial.
* A composite primary key must be represented and mapped as an embeddable class (EmbeddedId Annotation).

We shall explore the concept behind this composite primary keys in the next article.

# **Value object & Entity object in Hibernate mapping**

So far we have learnt how to write a model object and we learnt to annotate a model object so that hibernate create a table for us, then we learnt how to [save](http://www.simplecodestuffs.com/hibernate-4-hello-world-example-in-eclipse-using-annotation/), [retrieve](http://www.simplecodestuffs.com/example-on-hibernate-select-query/), [update](http://www.simplecodestuffs.com/hibernate-update-query-example/) and [delete](http://www.simplecodestuffs.com/hibernate-delete-query-example/) the entity object in the database.  
   
  
   
Still now we have created the entity class based on the assumption that each member variable inside the entity class will have a single column in database.  
   
In Above table the type of fields of the class STUDENT CLASS is as below.

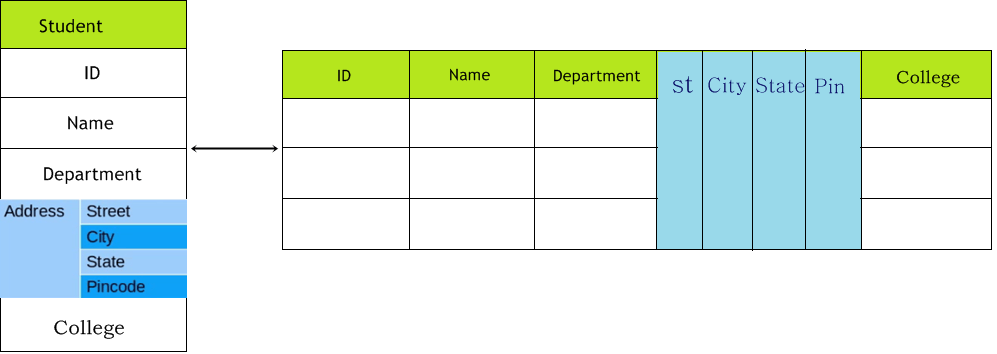
              ID            is INTEGER type  
              Name          is VARCHAR type  
              DEPARTMENT    is VARCHAR type  
              ADDRESS       is VARCHAR type  
              COLLEGE       is VARCHAR type

Now what if one of these member variable is an object instead of a simple data type. For example  
   
  
   
Here STUDENT CLASS have a field Address object, this means in the database there are no meaning for Address object, as this Address object in turn have four other fields like.

1. Street  
2. City  
3. State  
4. Pin code

Now the question is how can we save this address object into the database.

#### Value object

One way of saving the Address Object is to treat the member variable of address objects as the member variable of Student object itself.  
   
  
   
Here without having a Student object the use of Address object doesn’t make any sense. So the purpose of address object is just provide value to the Student object; And this is what differentiate an Entity object from a Value object.

##### **So what the heck is Value object?**

Entity object is the one which have meaning on its own, where a value object has no meaning on its own, the value object belongs to an entity instance and its persistent state is embedded in the table row of the owning entity.  
In the above example the Student object represents an Entity object and Address object represents a Value object.

##### **Long story short here:**

Entity Object: Has its own database identity (Student table)  
Value Object: Doesn’t not have its own database identity  
   
Now let’s create an class for Address (Value object)

**package** entity;  
  
**import** javax.persistence.**Column**;  
**import** javax.persistence.**Embeddable**;  
  
@Embeddable  
**public** **class** **Address** {  
        @Column(name = "STREET\_NAME")  
        **private** **String** street;  
        @Column(name = "CITY\_NAME")  
        **private** **String** city;  
        @Column(name = "STATE\_NAME")  
        **private** **String** state;  
        @Column(name = "PIN\_CODE")  
        **private** **String** pincode;  
          
        // Create getters and setters  
}

Now in order to make this class of value type, and in order to tell hibernate not to create a separate table for address class, I have used @Embeddable annotation in the Address class, Also we need to use @Embedded annotation in the member variable of Address object in Student entity class as shown below.

**package** entity;  
  
**import** javax.persistence.**Column**;  
**import** javax.persistence.**Embedded**;  
**import** javax.persistence.**Entity**;  
**import** javax.persistence.**GeneratedValue**;  
**import** javax.persistence.**Id**;  
**import** javax.persistence.**Table**;  
  
@Entity  
@Table(name = "STUDENT")  
**public** **class** **Student** {  
  
        @Id  
        @GeneratedValue  
        @Column(name = "ID")  
        **private** **int** id;  
  
        @Column(name = "NAME")  
        **private** **String** name;  
  
        @Column(name = "DEPARTMENT")  
        **private** **String** department;  
  
        @Column(name = "COLLEGE")  
        **private** **String** college;  
  
        // For value type object  
        @Embedded  
        **private** **Address** address;  
  
       // Getters and Setters  
}

#### Hibernate Configuration file

File: hibernate.cfg.xml

<hibernate-configuration>  
<session-factory>  
        <!-- Database connection settings -->  
        <property name="hibernate.connection.driver\_class">  
                 oracle.jdbc.driver.OracleDriver  
        </property>  
        <property name="hibernate.connection.username">system</property>  
        <property name="hibernate.connection.password">admin</property>  
        <property name="hibernate.connection.url">  
                jdbc:oracle:thin:@127.0.0.1:1521:XE  
        </property>  
  
        <!-- SQL dialect -->  
        <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>  
  
        <!-- Echo all executed SQL to sysout -->  
        <property name="show\_sql">true</property>  
  
        <!-- Drop and re-create the database schema on startup -->  
        <property name="hibernate.hbm2ddl.auto">create</property>  
        <!-- Map Entity Class -->  
    <mapping class="entity.Student"></mapping>  
  
</session-factory>  
  
</hibernate-configuration>

#### HibernateTest class

Create the Main class to run the example.

**package** util;  
  
**import** org.hibernate.**Session**;  
**import** org.hibernate.**SessionFactory**;  
**import** org.hibernate.**Transaction**;  
**import** org.hibernate.boot.registry.**StandardServiceRegistryBuilder**;  
**import** org.hibernate.cfg.**Configuration**;  
**import** org.hibernate.service.**ServiceRegistry**;  
  
**import** entity.**Address**;  
**import** entity.**Student**;  
  
**public** **class** **HibernateUtil** {  
  
        **public** **static** **void** main(**String**[] args) {  
  
                **Configuration** cf = **new** **Configuration**().configure("hibernate.cfg.xml");  
  
                **StandardServiceRegistryBuilder** srb =   
                         **new** **StandardServiceRegistryBuilder**();  
                srb.applySettings(cf.getProperties());  
                **ServiceRegistry** sr = srb.build();  
                **SessionFactory** sf = cf.buildSessionFactory(sr);  
  
                **Session** session = sf.openSession();  
                **Transaction** tx = session.beginTransaction();  
  
                **Student** student = **new** **Student**();   
  
                student.setName("Lahir Nisha");  
                student.setDepartment("ECE");  
                student.setCollege("SKCET");  
  
                **Address** address1 = **new** **Address**();  
                address1.setStreet("Race cource");  
                address1.setCity("Coimbatore");  
                address1.setState("Tamilnadu");  
                address1.setPincode("642001");  
                student.setAddress(address1);  
                session.save(student);    
  
                tx.commit();   
                session.close();  
                sf.close();  
        }  
}

#### Run it – Eclipse Console

Vlue object

#### Database Output

Vlue object saving

#### @AttributeOverride annotation:

So far we have seen that an Entity Type Object Student has a Value Object(or Embeddable Object ) ADDRESS with corresponding fields name street, city, pin-code and state save to the database table STUDENT with value object’s column name (CITY\_NAME, PIN\_CODE, STATE\_NAME, STREET\_NAME).  
   
Suppose in the above scenario if a Student have two addresses like Local Address and Permanent Address then how to manage the column names of these two value objects in the database table STUDENT.  
   
In order to overcome this problem, we have to override the Attribute name of the Value objects. Hibernate has provided @AttributeOverride annotation for this purpose.  
   
Example:-

##### **Student.java**

**package** entity;  
  
**import** javax.persistence.**AttributeOverride**;  
**import** javax.persistence.**AttributeOverrides**;  
**import** javax.persistence.**Column**;  
**import** javax.persistence.**Embedded**;  
**import** javax.persistence.**Entity**;  
**import** javax.persistence.**GeneratedValue**;  
**import** javax.persistence.**Id**;  
**import** javax.persistence.**Table**;  
  
@Entity  
@Table(name = "STUDENT")  
**public** **class** **Student** {  
  
        @Id  
        @GeneratedValue  
        @Column(name = "ID")  
        **private** **int** id;  
  
        @Column(name = "NAME")  
        **private** **String** name;  
  
        @Column(name = "DEPARTMENT")  
        **private** **String** department;  
  
        @Column(name = "COLLEGE")  
        **private** **String** college;  
  
        @Column(name = "ADDRESS")  
        @Embedded  
        @AttributeOverrides({  
           @AttributeOverride(name = "street", column = @Column(name = "HOME\_STREET\_NAME")),  
           @AttributeOverride(name = "city", column = @Column(name = "HOME\_CITY\_NAME")),  
           @AttributeOverride(name = "state", column = @Column(name = "HOME\_STATE\_NAME")),  
           @AttributeOverride(name = "pincode", column = @Column(name = "HOME\_PIN\_CODE")) })  
        **private** **Address** homeAddress;  
  
        @Embedded  
        **private** **Address** permanentAddress;  
        // Create getters and setters   
}

#### HibernateTest class

Create the Main class to run the example.

**package** util;  
  
**import** org.hibernate.**Session**;  
**import** org.hibernate.**SessionFactory**;  
**import** org.hibernate.**Transaction**;  
**import** org.hibernate.boot.registry.**StandardServiceRegistryBuilder**;  
**import** org.hibernate.cfg.**Configuration**;  
**import** org.hibernate.service.**ServiceRegistry**;  
  
**import** entity.**Address**;  
**import** entity.**Student**;  
  
**public** **class** **HibernateUtil** {  
  
        **public** **static** **void** main(**String**[] args) {  
  
                **Configuration** cf = **new** **Configuration**().configure("hibernate.cfg.xml");  
  
                **StandardServiceRegistryBuilder** srb   
                             = **new** **StandardServiceRegistryBuilder**();  
                srb.applySettings(cf.getProperties());  
                **ServiceRegistry** sr = srb.build();  
                **SessionFactory** sf = cf.buildSessionFactory(sr);  
  
                **Session** session = sf.openSession();  
                **Transaction** tx = session.beginTransaction();  
                  
                **Address** homeAddress = **new** **Address**();   
                homeAddress.setStreet("Race cource");  
                homeAddress.setCity("Coimbatore");  
                homeAddress.setState("Tamilnadu");  
                homeAddress.setPincode("642001");  
                  
                **Address** permanantAddress = **new** **Address**();  
                permanantAddress.setStreet("Besant nagar");  
                permanantAddress.setCity("Chennai");  
                permanantAddress.setState("Tamilnadu");  
                permanantAddress.setPincode("600001");  
                  
                **Student** student = **new** **Student**();   
  
                student.setName("Lahir Nisha");  
                student.setDepartment("ECE");  
                student.setCollege("SKCET");  
                student.setPermanentAddress(permanantAddress);  
                student.setHomeAddress(homeAddress);  
  
                session.save(student);   
          
                tx.commit();   
                session.close();  
                sf.close();  
        }  
}

#### Eclipse console after running this program

   
Hibernate: insert into STUDENT (COLLEGE, DEPARTMENT, HOME\_CITY\_NAME, HOME\_PIN\_CODE, HOME\_STATE\_NAME, HOME\_STREET\_NAME, NAME, CITY\_NAME, PIN\_CODE, STATE\_NAME, STREET\_NAME, ID) values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)  
   
In our next article we shall learn about [Saving Collections in Hibernate](http://www.simplecodestuffs.com/saving-collections-in-hibernate/)

# **Saving Collections in Hibernate**

In our last Hibernate article, we learned about [Embedding value object into an entity class](http://www.simplecodestuffs.com/value-object-entity-object-in-hibernate-mapping/). In this article we’re going to focus on saving a Collection in Hibernate.  
   
Consider a scenario where the student have lots of address. So in this case we need to create more no of embedded object in the student class, which is so tedious at certain point. In such case we can use Collection to solve this problem. This collection can be a list, set, map, collection, sorted set, sorted map. Hibernate provides the facility to persist the collections object via@ElementCollection annotation.

#### Entity Class

**package** entity;  
  
**import** java.util.**ArrayList**;  
**import** java.util.**Collection**;  
  
**import** javax.persistence.**Column**;  
**import** javax.persistence.**ElementCollection**;  
**import** javax.persistence.**Entity**;  
**import** javax.persistence.**GeneratedValue**;  
**import** javax.persistence.**Id**;  
**import** javax.persistence.**Table**;  
  
@Entity  
@Table(name = "STUDENT")  
**public** **class** **Student** {  
  
        @Id  
        @GeneratedValue  
        @Column(name = "ID")  
        **private** **int** id;  
  
        @Column(name = "NAME")  
        **private** **String** name;  
  
        @Column(name = "DEPARTMENT")  
        **private** **String** department;  
  
        @Column(name = "COLLEGE")  
        **private** **String** college;  
  
        @ElementCollection  
        **private** **Collection**<**Address**> lisOfAddress = **new** **ArrayList**<**Address**>();  
  
        // Create Getters and Setters  
}

##### **@ElementCollection:**

Defines a collection of instances of an embeddable class, this annotation must be specified if the collection is to be mapped by means of a collection table.

**package** entity;  
  
**import** javax.persistence.**Column**;  
**import** javax.persistence.**Embeddable**;  
  
@Embeddable  
**public** **class** **Address** {  
        @Column(name = "STREET\_NAME")  
        **private** **String** street;  
        @Column(name = "CITY\_NAME")  
        **private** **String** city;  
        @Column(name = "STATE\_NAME")  
        **private** **String** state;  
        @Column(name = "PIN\_CODE")  
        **private** **String** pincode;  
          
        // Create Getters and Setters  
}

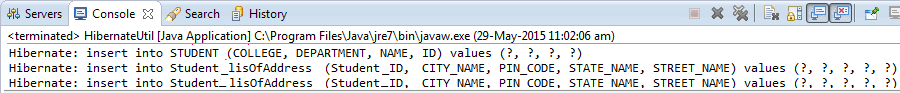
hibernate.cfg.xml will be the same as in previous article.

#### HibernateTest class

Create the Main class to run the example.

**package** util;  
  
**import** org.hibernate.**Session**;  
**import** org.hibernate.**SessionFactory**;  
**import** org.hibernate.**Transaction**;  
**import** org.hibernate.boot.registry.**StandardServiceRegistryBuilder**;  
**import** org.hibernate.cfg.**Configuration**;  
**import** org.hibernate.service.**ServiceRegistry**;  
  
**import** entity.**Address**;  
**import** entity.**Student**;  
  
**public** **class** **HibernateUtil** {  
  
        **public** **static** **void** main(**String**[] args) {  
  
                **Configuration** cf = **new** **Configuration**().configure("hibernate.cfg.xml");  
  
                **StandardServiceRegistryBuilder** srb =   
                               **new** **StandardServiceRegistryBuilder**();  
                srb.applySettings(cf.getProperties());  
                **ServiceRegistry** sr = srb.build();  
                **SessionFactory** sf = cf.buildSessionFactory(sr);  
  
                **Session** session = sf.openSession();  
                **Transaction** tx = session.beginTransaction();  
                  
                **Address** address1 = **new** **Address**();   
                address1.setStreet("Race cource");  
                address1.setCity("Coimbatore");  
                address1.setState("Tamilnadu");  
                address1.setPincode("642001");  
                  
                **Address** address2 = **new** **Address**();  
                address2.setStreet("Besant nagar");  
                address2.setCity("Chennai");  
                address2.setState("Tamilnadu");  
                address2.setPincode("600001");  
                  
                **Student** student = **new** **Student**();   
  
                student.setName("Lahir Nisha");  
                student.setDepartment("ECE");  
                student.setCollege("SKCET");  
                  
                //adding addresses object to the list of address  
                student.getLisOfAddress().add(address1);  
                student.getLisOfAddress().add(address2);  
  
                session.save(student);   
          
                tx.commit();   
                session.close();  
                sf.close();  
        }  
}

#### Rut it – Eclipse Console

  
   
When you run the above program two tables are created in database, first table is for Student entity “STUDENT” and second table is for the embeddable object (Address) – “Student\_lisOfAddress” (The default table name for this table gets generated based on – Entity class name\_Name of the embeddable object in the entity class).  
   
Here the ‘Student\_ID’ column in ‘Student\_lisOfAddress’ is the foreign key reference of Student tables ‘ID’. In [next article](http://www.simplecodestuffs.com/configuring-adding-primary-key-to-collections-in-hibernate/) we shall learn to override the default collection class table name and to add primary key to it.