replicate

saveOrUpdate

merge

persist

refresh

• save()

• update()

• saveOrUpdate()

• saveOrUpdateCopy()

• merge()

• persist()

http://stackoverflow.com/questions/2573739/what-are-detached-persistance-and-transient-object-in-hibernate

http://www.hibernate-training-guide.com/detached-state.html

http://www.mastishare.com/video/10451/indian-desi-babes-doing-lesbian-sex-in-hostel-room

http://www.java4s.com/hibernate/life-cycle-of-pojo-class-objects-in-the-hibernate/ \*\*\*

The main function of the <tt>Session</tt> is to offer create, read and delete operations

\* for instances of mapped entity classes. Instances may exist in one of three states:<br>

\* <br>

\* <i>transient:</i> never persistent, not associated with any <tt>Session</tt><br>

\* <i>persistent:</i> associated with a unique <tt>Session</tt><br>

\* <i>detached:</i> previously persistent, not associated with any <tt>Session</tt><br>

\* <br>

**Interface org.hibernate.Session Methods:**

1. **public** Serializable save(Object object) **throws** HibernateException;
   * This method first assigns the PK value to the object (based on <generator> element) & then it insert the Object values into the corresponding DB table columns
   * This method return the PK value
2. **public** **void** update(Object object) **throws** HibernateException;

* This method update the values of the DB row with values present in the given object by making use of PK value present in the Object
* This method does not return any value

1. **public** **void** saveOrUpdate(Object object) **throws** HibernateException;

* Thismethod either saves or updates the DB row with the values present in the Object
* This method first checks whether DB record exists with the given PK, if exists then then it updates & if it does not exists the it inserts

1. **public** Object merge(Object object) **throws** HibernateException;
2. **public** **void** persist(Object object) **throws** HibernateException;
3. **public** **void** delete(Object object) **throws** HibernateException;

/\*\*

\* Return the persistent instance of the given entity class with the given identifier,

\* assuming that the instance exists. This method might return a proxied instance that

\* is initialized on-demand, when a non-identifier method is accessed.

\* <br><br>

\* You should not use this method to determine if an instance exists (use <tt>get()</tt>

\* instead). Use this only to retrieve an instance that you assume exists, where non-existence

\* would be an actual error.

\*

\* **@param** theClass a persistent class

\* **@param** id a valid identifier of an existing persistent instance of the class

\* **@return** the persistent instance or proxy

\* **@throws** HibernateException

\*/

1. **public** Object load(Class theClass, Serializable id) **throws** HibernateException;

/\*\*

\* Return the persistent instance of the given entity class with the given identifier,

\* or null if there is no such persistent instance. (If the instance is already associated

\* with the session, return that instance. This method never returns an uninitialized instance.)

\*

\* **@param** clazz a persistent class

\* **@param** id an identifier

\* **@return** a persistent instance or null

\* **@throws** HibernateException

\*/

**public** Object get(Class clazz, Serializable id) **throws** HibernateException;

**When to use get? When to use load?**

* The load method may be the method of choice if you know and absolutely sure that the record you are searching, exists in the database with the given primary key.
* If you don't know for sure that the record exists in DB with the given primary key then you can use the get method and do a NULL check on the instance that gets returned from the get method call.

/\*\*

\* Re-read the state of the given instance from the underlying database. It is

\* inadvisable to use this to implement long-running sessions that span many

\* business tasks. This method is, however, useful in certain special circumstances.

\* For example

\* <ul>

\* <li>where a database trigger alters the object state upon insert or update

\* <li>after executing direct SQL (eg. a mass update) in the same session

\* <li>after inserting a <tt>Blob</tt> or <tt>Clob</tt>

\* </ul>

\*

\* **@param** object a persistent or detached instance

\* **@throws** HibernateException

\*/

**public** **void** refresh(Object object) **throws** HibernateException;

/\*\*

\* Begin a unit of work and return the associated <tt>Transaction</tt> object.

\* If a new underlying transaction is required, begin the transaction. Otherwise

\* continue the new work in the context of the existing underlying transaction.

\* The class of the returned <tt>Transaction</tt> object is determined by the

\* property <tt>hibernate.transaction\_factory</tt>.

\*

\* **@return** a Transaction instance

\* **@throws** HibernateException

\* **@see** Transaction

\*/

**public** Transaction beginTransaction() **throws** HibernateException;

/\*\*

\* Create a new instance of <tt>Query</tt> for the given HQL query string.

\*

\* **@param** queryString a HQL query

\* **@return** Query

\* **@throws** HibernateException

\*/

**public** Query createQuery(String queryString) **throws** HibernateException;

/\*\*

\* Get the <tt>Transaction</tt> instance associated with this session.

\* The class of the returned <tt>Transaction</tt> object is determined by the

\* property <tt>hibernate.transaction\_factory</tt>.

\*

\* **@return** a Transaction instance

\* **@throws** HibernateException

\* **@see** Transaction

\*/

**public** Transaction getTransaction();

/\*\*

\* Force this session to flush. Must be called at the end of a

\* unit of work, before committing the transaction and closing the

\* session (depending on {@link #setFlushMode flush-mode},

\* {@link Transaction#commit()} calls this method).

\* <p/>

\* <i>Flushing</i> is the process of synchronizing the underlying persistent

\* store with persistable state held in memory.

\*

\* **@throws** HibernateException Indicates problems flushing the session or

\* talking to the database.

\*/

**public** **void** flush() **throws** HibernateException;

/\*\*

\* End the session by releasing the JDBC connection and cleaning up. It is

\* not strictly necessary to close the session but you must at least

\* {@link #disconnect()} it.

\*

\* **@return** the connection provided by the application or null.

\* **@throws** HibernateException Indicates problems cleaning up.

\*/

**public** Connection close() **throws** HibernateException;

**public** **class** Hibernate\_SaveOrUpdate\_Transaction\_Example\_2

{

**public** **static** **void** main(String[] args)

{

Session session = **null**;

Transaction txn = **null**;

**try**

{

SessionFactory factory

= **new** Configuration().configure().buildSessionFactory();

session = factory.openSession();

txn = session.beginTransaction();

Students studentsObj = **new** Students();

studentsObj.setRegno(10);

studentsObj.setFirstName("Jhony");

studentsObj.setLastName("Walker");

System.*out*.println("Inserting the Record");

/\*

\* Session.save() :

\* It saves the given object into the Database & returns the

\* generated PK value

\*/

Integer regno = (Integer) session.save(studentsObj);

System.*out*.println("Inserted RegNo : "+regno);

/\*

\* Session.persist() :

\* It saves the given object into the Database & it doesn't

\* returns anything

\*/

session.persist(studentsObj);

/\*

\* Session.saveOrUpdate() :

\* This method run either INSERT or UPDATE SQL Command based upon

\* existence of object in database.

\* If persistence object already exists in database then Hibernate

\* will run UPDATE SQL Command

\* if there is no corresponding object exists in database then

\* Hibernate will run INSERT SQL Command

\* This method doesn't return anything

\*

\* Note :

\* 1. saveOrUpdate() involves an extra processing to find out

\* whether record already exists in table or not.

\* 2. If the Object passed to this method & the Object obtained by

\* Hibernate by Select Query are in sync then hibernate will not

\* run Update / Insert Command

\* 3. The behavior of this operation is different if we have

\* <generator class="increment" />

\* In this case hibernate checks whether the given object has the

\* PK or not. If it has then it will issue

\* the update statement & if it doesn't has then it will issue the

\* insert statement. If you have provided

\* the PK value which doesn't exists in DB then Hibernate throws

\* Exception

\*/

session.saveOrUpdate(studentsObj);

/\*

\* Session.merge() :

\* Operation is same as saveOrUpdate() method but it returns the

\* persisted Object

\*

\*/

Students studentsObj2 = (Students) session.merge(studentsObj);

System.*out*.println(studentsObj2);

session.flush();

System.*out*.println("Record got Inserted/Updated successfully");

//int i = 1/0; //Error Simulation

System.*out*.println("Commiting the transaction");

txn.commit();

}**catch**(Exception e){

System.*out*.println("Error Occurred hence rolling back the transaction");

txn.rollback();

e.printStackTrace();

}**finally**{

**if**(session!=**null**){

session.close();

}

}//End of Try-Catch Block

}//End of Main Method

}//End of Class

**Handling the DB Table default values in Hibernate**

If a table has any default values then declare those default values as bean property default values in Java Bean. Rest falls in place.

**private** **int** regno; //DB Table does not have the Default Value

**private** String firstName; //DB Table does not have the Default Value

**private** String middleName="NotAvilable";//DB Table Default Value is ‘NotAvilable’

**private** String lastName; //DB Table does not have the Default Value

**private** String password = "qwerty"; //DB Table Default Value is ‘qwerty’

**private** String isAdmin = "N"; //DB Table Default Value is ‘qwerty’

replicate

saveOrUpdate

merge

persist

refresh

• save()

• update()

• saveOrUpdate()

• saveOrUpdateCopy()

• merge()

• persist()

http://stackoverflow.com/questions/2573739/what-are-detached-persistance-and-transient-object-in-hibernate

http://www.hibernate-training-guide.com/detached-state.html

http://www.mastishare.com/video/10451/indian-desi-babes-doing-lesbian-sex-in-hostel-room

http://www.java4s.com/hibernate/life-cycle-of-pojo-class-objects-in-the-hibernate/ \*\*\*

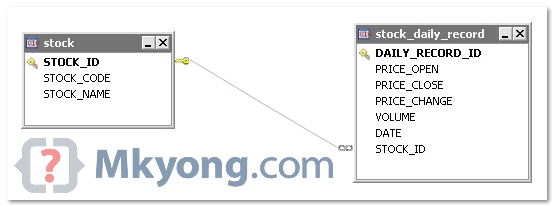
**PENDING ITEMS W.R.T Hibernate**

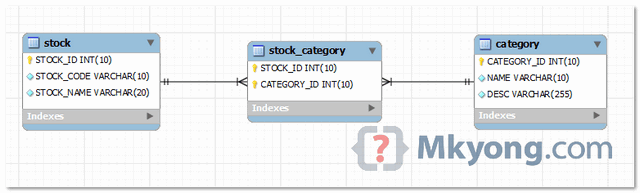
* **Difference between openSession() & getCurrentSession()**
* **Hibernate – How to handle PK & FK relationship.**
* **How To Call Stored Procedure In Hibernate**

<http://www.mkyong.com/hibernate/how-to-call-store-procedure-in-hibernate/>

* Using orders in Hibernate
* Study about org.hibernate.type.descriptor.java.UUIDTypeDescriptor.ToStringTransformer
* In Hibernate, only the “relationship owner” should maintain the relationship, and the “inverse” keyword is created to defines which side is the owner to maintain the relationship
* In short, inverse=”true” means this is the relationship owner, and inverse=”false” (default) means it’s not.
* About Hibernate Caching

<http://viralpatel.net/blogs/introduction-to-hibernate-framework-architecture/>





name: the name of the property.

class (optional - defaults to the property type determined by reflection): the name of the associated class.

cascade (optional): specifies which operations should be cascaded from the parent object to the associated object.

fetch (optional - defaults to select): chooses between outer-join fetching or sequential select fetching.

property-ref (optional): the name of a property of the associated class that is joined to the primary key of this class. If not specified, the primary key of the associated class is used.

access (optional - defaults to property): the strategy Hibernate uses for accessing the property value.

formula (optional): almost all one-to-one associations map to the primary key of the owning entity. If this is not the case, you can specify another column, columns or expression to join on using an SQL formula. See org.hibernate.test.onetooneformula for an example.

lazy (optional - defaults to proxy): by default, single point associations are proxied. lazy="no-proxy" specifies that the property should be fetched lazily when the instance variable is first accessed. It requires build-time bytecode instrumentation. lazy="false" specifies that the association will always be eagerly fetched. Note that if constrained="false", proxying is impossible and Hibernate will eagerly fetch the association.

entity-name (optional): the entity name of the associated class.

constrained (optional): specifies that a foreign key constraint on the primary key of the mapped table and references the table of the associated class. This option affects the order in which save() and delete() are cascaded, and determines whether the association can be proxied. It is also used by the schema export tool.

Managing Association:

1. Association is a relationship of Once Java Bean to another Java Bean. Usually DB design involves tables to have relationship with one another

2 In DB a table can have following relationships

Hibernate – Fetching Strategies Examples

http://www.mkyong.com/hibernate/hibernate-fetching-strategies-examples/

http://www.javalobby.org/java/forums/t20533.html

Hibernate Component Mapping Example

<http://www.mkyong.com/hibernate/hibernate-component-mapping-example/>

<https://forum.hibernate.org/viewtopic.php?p=2387456>

 The difference between a Hibernate load and a Hibernate get is a very common question, but the differences are important, although, at times, the difference between Hibernate load vs. get can be subtle.   
  
**Loading Entities with Hibernate**   
  
In an earlier tutorial, we looked at the hypothetical example where you had the primary key of an entity, and wanted to query the database and have Hibernate return the unique User instance associated with that primary key. The Haitian Voodoo required to perform that retrieval task looked something like this:

**Code:**

public static User retrieveFromId(int idValue) {  
  AnnotationConfiguration config =    
                       new   AnnotationConfiguration();  
  config.addAnnotatedClass(User.class);  
  SessionFactory factory;  
  factory = config.configure().buildSessionFactory();  
  Session session = factory.getCurrentSession();  
  session.beginTransaction();  
  String queryString = "from User where id = :id";  
  Query query = session.createQuery(queryString);  
  query.setInteger("id", idValue);  
  Object queryResult = query.uniqueResult();  
  User user = (User)queryResult;  
  session.getTransaction().commit();  
  System.out.print(user.getPassword());  
  return user;  
}

Taking a primary key, and using it to demonstrate variable injection and the retrieval of a unique entity was very androgologically sound, but the fact of the matter is, if you actually have the primary key of an entity, there is a much easier, or should I say "a couple of much easier ways" to retrieve the corresponding entity from the database. We'll demonstrate those ways by coding a few methods into a new class called the LoadRunner.  
  
**The LoadRunner Class**  
  
The LoadRunner class is going to have a runnable main method, and two static methods, named callGet and callLoad, which will be used to demonstrate the two different ways you can get at an entity given its associated primary key. For now, I've coded in all the redundant code we need to have each method connect to the database, create a session, and start a transaction. I know that all of this Hibernate plumbing code is getting repetitive; we'll factor it out into a HibernateUtil class soon enough.

**Code:**

package com.examscam;  
import org.hibernate.\*;   
  
import com.examscam.model.User;  
import org.hibernate.cfg.AnnotationConfiguration;  
public class LoadRunner {  
  public static void callLoad(){  
   AnnotationConfiguration config  
                    = new AnnotationConfiguration();  
   config.addAnnotatedClass(User.class);  
   SessionFactory factory=  
             config.configure().buildSessionFactory();  
   Session session = factory.getCurrentSession();  
   session.beginTransaction();  
/\*\*\*\*\* load code will go here \*\*\*\*\*/  
   session.getTransaction().commit();  
  }  
  public static void callGet() {  
   AnnotationConfiguration config  
                    = new AnnotationConfiguration();  
   config.addAnnotatedClass(User.class);  
   SessionFactory factory=  
             config.configure().buildSessionFactory();  
   Session session = factory.getCurrentSession();  
   session.beginTransaction();  
/\*\*\*\*\* get code will go here \*\*\*\*\*/  
   session.getTransaction().commit();  
  }  
  public static void main(String[] args) {  
    LoadRunner.callLoad();  
    LoadRunner.callGet();  
  }  
}

**The callGet Method**  
  
Within the session.beginTransaction() and the session.getTransaction().commit methods, we want to use the Hibernate Session to take a given primary key, and provide our program with the entity associated with that primary key in the database. It's all a pretty simple process. To get an entity from the underlying persistence store, all you have to do is call the Hibernate Session's get method, and provide two arguments: the Java class associated with the entity that your are retrieving, which in this case would be the User.class, and the actual primary key associated with the record. I know that I have a user in the database with a primary key of 1, so the number one will be used to test my methods.  
  
Here's all the code you need to resurrect a User instance from the database using the Hibernate Session's load method:

**Code:**

session.beginTransaction();  
User user = (User)session.get(User.class, new Long(1));  
System.out.println(user.getPassword());  
session.getTransaction().commit();

And that's about it! You simply call the load method of the Hibernate Session, provide the class type and the primary key as parameters, and then cast the object that is returned from Hibernate back into the appropriate Java type. From there, you can do just about anything with your JavaBean that you want. In this case, I simply print out the password of the User instance. It's all just so easy.  
  
Now, as I mentioned, there are two ways to pull an entity from the database, one of which is to use the get method of the Hibernate Session, and the other way is to use the load method. Compare and contrast the following code snippet that invokes the Session's load method to the code snippet that invoked the Session's get method. The difference is subtle, to say the least.

**Code:**

session.beginTransaction();  
User user = (User)session.load(User.class, new Long(1));  
System.out.println(user.getPassword());  
session.getTransaction().commit();

**Hibernate load vs. Hibernate get Methods**  
Well, if you were to compare the load and get methods of the Hibernate Session, you'd think that they looked pretty darned similar; and you'd be correct, but there are subtle and very important differences.  
  
First of all, the get method hits the database as soon as it is called. So, using the Hibernate Session's get method will always trigger a database hit. On the other hand, the load method only hits the database when a particular field of the entity is accessed. So, if we use the load method to retrieve an entity, but we never actually access any of the fields of that entity, we never actually hit the database. Pretty kewl, eh?  
  
Well, actually, as kewl as the load method might sound, it actually triggers more problems than it solves, and here's why. If you initialize a JavaBean instance with a load method call, you can only access the properties of that JavaBean, for the first time, within the transactional context in which it was initialized. If you try to access the various properties of the JavaBean after the transaction that loaded it has been committed, you'll get an exception, a LazyInitializationException, as Hibernate no longer has a valid transactional context to use to hit the database.  
  
**So, while this code will work just fine?..**

**Code:**

session.beginTransaction();  
User user=(User)session.load(User.class, new Long(1));  
System.out.println(user.getPassword());  
session.getTransaction().commit();

**.. this code will fail ?..**

**Code:**

session.beginTransaction();  
User user=(User)session.load(User.class, new Long(1));  
session.getTransaction().commit();  
System.out.println(user.getPassword());

.. and generate the following error, telling you that since the transaction was committed, there was no valid Session in which a read transaction against the database could be issued:  
  
**org.hibernate.LazyInitializationException:**  
*could not initialize proxy - no Session*  
  
So, the big thing to take away from this is that with the load method, you can't really use your loaded JavaBeans after the transaction has been committed, whereas, with the get method you can, because all of the various properties of a JavaBean retrieved through the get method are initialized right away.  
Loading Non-Existent Records  
  
An important scenario under which you need to contrast the load and get methods of the Hibernate Session has to do with what happens when you provide a primary key that doesn't actually exist in the database. Well, with the get method, you are simply returned a null object, which is no big deal.  
  
With the load method, there's also no initial problem when you provide an invalid primary key to the method. From what you can tell, Hibernate appears to hand you back a valid, non-null instance of the class in which you are interested. However, the problems start when you actually try to access a property of that instance - that's where you run into trouble.  
  
Remember how I said the load method doesn't hit the database until a property of the bean is requested? Well, if you've provided a primary key that doesn't exist in your database to the load method, when it does go to the database for the first time, it won't be able to find the non-existent, associated record, and your code will cough up big time. In fact, looking up a field based upon a non-existent primary key with the Hibernate Session's load method triggers the following error:  
  
**org.hibernate.ObjectNotFoundException:**  
No row with the given identifier exists: [User#123]

**Quote:**

public Object get (Class clazz, Serializable id) throws HibernateException  
--Return the persistent instance of the given entity class with the given identifier, or null if there is no such persistent instance. (If the instance is already associated with the session, return that instance or proxy.)

**Quote:**

public Object load (Class theClass,Serializable id) throws HibernateException  
--Return the persistent instance of the given entity class with the given identifier, assuming that the instance exists. You should not use this 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.

-Regurgitated Hibernate API JavaDoc

**Code:**

package com.examscam;  
import org.hibernate.\*; import com.examscam.model.User;  
import org.hibernate.cfg.AnnotationConfiguration;  
public class LoadRunner {  
  
  public static void main(String[] args) {  
    LoadRunner.callLoad();  LoadRunner.callGet();  
  }  
  public static void callLoad(){  
   AnnotationConfiguration config =   
                  new AnnotationConfiguration();  
   config.addAnnotatedClass(User.class);  
   SessionFactory factory=     
            config.configure().buildSessionFactory();  
   Session session = factory.getCurrentSession();  
   session.beginTransaction();  
   try {  
     User user=  
       (User)session.load(User.class,new Long(1));  
    System.out.println(user.getPassword ());  
   } catch (ObjectNotFoundException e) {  
     e.printStackTrace();  
   }  
   session.getTransaction().commit();  
     
/\* System.out.println(user.getPassword());   
   This would fail!!! \*/  
     
  }  
  public static void callGet() {  
    AnnotationConfiguration config =   
                  new AnnotationConfiguration();  
    config.addAnnotatedClass(User.class);  
    SessionFactory factory=     
            config.configure().buildSessionFactory();  
    Session session = factory.getCurrentSession();  
   session.beginTransaction();  
   User user=  
      (User)session.get(User.class,new Long(1));  
    System.out.println(user.getPassword());  
    session.getTransaction().commit();  
    /\* no problem!!!\*/  
    System.out.println(user.getPassword ());   
  }  
}

<http://viralpatel.net/blogs/hibernate-one-to-many-xml-mapping-tutorial/>

<http://www.javabeat.net/2008/09/many-to-one-relationship-in-hibernate-mappings-example/>

<http://www.dineshonjava.com/p/hibernate-many-to-one-mapping-tutorial.html#.UPlI6B04vPY>

 A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

 Configuration cfg = new Configuration()

    .addResource("Employee.hbm.xml")

    .addResource("Department.hbm.xml");

Configuration cfg = new Configuration()

    .addClass(net.viralpatel.hibernate.Employee.class)

    .addClass(net.viralpatel.hibernate.Department.class);

 Configuration cfg = new Configuration()

    .addClass(net.viralpatel.hibernate.Employee.class)

    .addClass(net.viralpatel.hibernate.Department.class)

    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")

    .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")

    .setProperty("hibernate.order\_updates", "true");

[http://blog.sencide.com/2011/03/hibernate-tutorial-for-beginners.html](https://owa.hmhpub.com/owa/redir.aspx?C=dtSvXqNrX0-KM5zG4QJbMRJpQOzlz88IT3Ns7ZFJbhuOKbBB7rfjhP0DVx_mE53HnXIXJCNuM68.&URL=http%3a%2f%2fblog.sencide.com%2f2011%2f03%2fhibernate-tutorial-for-beginners.html)

[http://www.visualbuilder.com/java/hibernate/tutorial/introduction-to-java-hibernate/](https://owa.hmhpub.com/owa/redir.aspx?C=dtSvXqNrX0-KM5zG4QJbMRJpQOzlz88IT3Ns7ZFJbhuOKbBB7rfjhP0DVx_mE53HnXIXJCNuM68.&URL=http%3a%2f%2fwww.visualbuilder.com%2fjava%2fhibernate%2ftutorial%2fintroduction-to-java-hibernate%2f)

<set name=*"phoneNumbers"* table=*"phone\_numbers"* cascade=*"all"* inverse=*"true"*>

<key column=*"regno"* />

<one-to-many class=*"com.jspiders.hibernate.beans.PhoneNumber"* />

</set>

When INVERSE was false ( i.e. Student was the owner of relationship ) the intermediary table STUD\_PHONE was updated. When inverse is true, Phone is owner of relationship, so even though a new student was created, the intermediary table was not updated.

So in case of relation of two entities, "cascade" affects other entity table and "inverse" affects intermediary table. So their effect is independent.

**Lazy Keyword**

* Lazy keyword decides whether to load child objects while loading the Parent Object.
* Lazy = true, it’s a default means not to load child. This makes sure that the child objects are not loaded unless they are explicitly invoked in the application by calling get method on parent.
* Lazy=false, hibernate will load the child when parent is loaded from the database.
* If we try to access the child objects lazily (Lazy = true) after closing the session, then we will get the LazyInitializationException.

**Difference between Lazy & Fetch:**

* The lazy attribute tells hibernate when to get the children
* The fetch attribute tells hibernate how to get the children
* But fetch="select" & lazy=”true” Both meaning the same

**not-null Keyword :**

* not-null=”true”, specifies that the columns cannot hold NULL values
* not-null=”false”, specifies that the columns can hold NULL values

**1.** [**Serializable**](eclipse-javadoc:%E2%98%82=Hibernate/D:%5C/Praveen%5C/tools%5C/ReferenceJars%5C/hibernate%203.6%5C/hibernate3.jar%3Corg.hibernate(Session.class%E2%98%83Session~save~Ljava.lang.Object;%E2%98%82Serializable)[**Session**](eclipse-javadoc:%E2%98%82=Hibernate/D:%5C/Praveen%5C/tools%5C/ReferenceJars%5C/hibernate%203.6%5C/hibernate3.jar%3Corg.hibernate(Session.class%E2%98%83Session)**.save(**[**Object**](eclipse-javadoc:%E2%98%82=Hibernate/D:%5C/Praveen%5C/tools%5C/ReferenceJars%5C/hibernate%203.6%5C/hibernate3.jar%3Corg.hibernate(Session.class%E2%98%83Session~save~Ljava.lang.Object;%E2%98%82Object) **objRef) method :**  
It saves the given object into the Database & returns the generated PK value

Main *difference between save and saveOrUpdate method* is that save() generates a **new identifier** and INSERT

record into [database](http://javarevisited.blogspot.sg/2011/10/selct-command-sql-query-example.html) while saveOrUpdate can either INSERT or UPDATE based upon existence of record. Clearly saveOrUpdate is more flexible in terms of use but it involves an extra processing to find out whether record already exists in table or not. In summary save() method saves records into database by INSERT SQL query, Generates a new identifier and return the [Serializable](http://javarevisited.blogspot.sg/2011/04/top-10-java-serialization-interview.html) identifier back. On the other hand saveOrUpdate() method either INSERT or UPDATE based upon existence of object in database. If persistence object already exists in database then UPDATE SQL will [execute](http://javarevisited.blogspot.sg/2012/03/how-to-create-and-execute-jar-file-in.html) and if there is no corresponding object in database than INSERT will run.  
  
These were some **differences between save, saveOrUpdate and persist method of Hibernate**. All three method are related to saving Object into database but there behavior are quite different. Knowledge of save, persist and saveOrUpdate not only helps to decide better use of Hibernate API but also help you to do well in Hibernate interviews.

First difference between save and persist is there return type. Similar to save method persist also INSERT records into database but **return type of persist is void** while return type of save is [Serializable](http://javarevisited.blogspot.sg/2012/01/serializable-externalizable-in-java.html) object.

Actually the difference between hibernate save() and persist() methods is depends on generator class we are using.

* If our generator class is assigned, then there is no difference between save() and persist() methods. Because generator ‘assigned’ means, as a programmer we need to give the primary key value to save in the database right [ Hope you know this generators concept ]
* In case of other than assigned generator class, suppose if our generator class name is Increment means hibernate it self will assign the primary key id value into the database right [ other than assigned generator, hibernate only used to take care the primary key id value remember ], so in this case if we call save() or persist() method then it will insert the record into the database normally

But hear thing is, save() method can return that primary key id value which is generated by hibernate and we can see it by

long s = session.save(k);

In this same case, persist() will never give any value back to the client, hope you are clear.

**Hibernates saveOrUpdate() & merge() Method :**

This method internally calls

1. Sessions save() method if there is a record with given PK does not exists in DB
2. Session update() method if there is a record with given PK exists in DB

**public** **class** Hibernate\_SaveOrUpdate\_Transaction\_Example

{

**public** **static** **void** main(String[] args)

{

Session session = **null**;

Transaction txn = **null**;

**try**

{

SessionFactory factory

= **new** Configuration().configure().buildSessionFactory();

session = factory.openSession();

txn = session.beginTransaction();

Students studentsObj = **new** Students();

studentsObj.setRegno(3);

studentsObj.setFirstName("AAA");

studentsObj.setMiddleName("BBB");

studentsObj.setLastName("CCC");

System.*out*.println("Inserting the Record");

session.saveOrUpdate(studentsObj);

session.flush();

System.*out*.println("Record got Inserted/Updated successfully");

//int i = 1/0; //Error Simulation

System.*out*.println("Commiting the transaction");

txn.commit();

}**catch**(Exception e){

System.*out*.println("Error Occurred hence rolling back the transaction");

txn.rollback();

e.printStackTrace();

}**finally**{

**if**(session!=**null**){

session.close();

}

}//End of Try-Catch Block

}//End of Main Method

}//End of Class

<http://www.java4s.com/hibernate/generators-in-hibernate/>

<http://www.stevideter.com/2008/12/07/saveorupdate-versus-merge-in-hibernate/>

<http://javarevisited.blogspot.in/2012/09/difference-hibernate-save-vs-persist-and-saveOrUpdate.html>

<http://www.roseindia.net/answers/viewqa/Hibernate/12197-Hibernate-session.refresh()-method.html>

<http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/associations.html#assoc-unidirectional-join-m21>