**Hibernate Object Lifecycle**

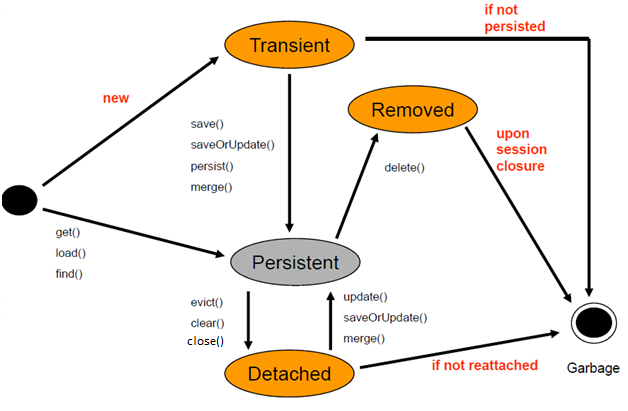
**Hibernate considers objects it canmanage to always be in one of fourstates**

– Transient

– Persistent

– Removed

– Detached



**Transient State**

• **All objects start off in the transient**

**state**

– Account account = new Account();

• account is a transient object

• **Hibernate is not aware of the object**

**Instance**

• **Not related to database row**

– No value for accounted

• **Garbage collected when no longer**

**referenced by any other objects**

**Persistent State**

• **Hibernate is aware of, and managing, the object**

• **Has a database id**

– Already existing object retrieved from the database

– Formerly transient object about to be saved

• **This is the only state where objects are saved to the**

**database**

– Modifications made in other states are NOT saved to the database while

the object remains in that state

– Changes to objects in a persistent state are automatically saved to the

database without invoking session persistence methods

• **Objects are made persistent through calls against the**

**Hibernate session**

– session.save(account); – session.lock(account);

– session.update(account); – session.merge(account);

**Removed State**

• **A previously persistent object that is**

**deleted from the database**

– session.delete(account);

• **Java instance may still exist, but it is**

**ignored by Hibernate**

– Any changes made to the object are not saved to

the database

– Picked up for garbage collection once it falls out

of scope

• Hibernate does *not* null-out the in-memory

object

**Session session = SessionFactory.getCurrentSession();**

**// retrieve account with id 1. account is returned in a ‘persistent’ state**

**Account account = session.get(Account.class, 1);**

**// transition to the ‘removed’ state. Hibernate deletes the**

**// database record, and no longer manages the object**

**session.delete(account);**

**// modification is ignored by Hibernate since it is in the ‘removed’ state**

**account.setBalance(500);**

**// commit the transaction**

**session.getTransaction().commit();**

**// notice the Java object is still alive, though deleted from the database.**

**// stays alive until developer sets to null, or goes out of scope**

**account.setBalance(1000);**

**Detached State**

• **A persistent object that is still referenced**

**after closure of the active session**

– session.close() changes object’s state from persisted to

Detached

• **Still represents a valid row in the database**

• **No longer managed by Hibernate**

– Changes made to detached objects are not saved to the

database while object remains in the detached state

– Can be reattached, returning it to the persistent state and

causing it to save its state to the database

• update();

• merge();

• lock(); // reattaches, but does not save state

**Session session1 = SessionFactory.getCurrentSession();**

**// retrieve account with id 1. account is returned in a ‘persistent’ state**

**Account account = session1.get(Account.class, 1);**

**// transition to the ‘detached’ state. Hibernate no longer manages the object**

**session1.close();**

**// modification is ignored by Hibernate since it is in the ‘detached’**

**// state, but the account still represents a row in the database**

**account.setBalance(500);**

**// re-attach the object to an open session, returning it to the ‘persistent’**

**// state and allowing its changes to be saved to the database**

**Session session2 = SessionFactory.getCurrentSession();**

**session2.update(account);**

**// commit the transaction**

**session2.getTransaction().commit();**

**Saving Changes to the Database**

• **Session methods do NOT save changes to**

**the database**

– save();

– update();

– delete();

• **These methods actually SCHEDULE**

**changes to be made to the database**

• **Hibernate collects SQL statements to be**

**issued**

• **Statements are later *flushed* to the database**

– Once submitted, modifications to the database are not

permanent until a commit is issued

• session.getTransaction().commit();

**Session API –**

**Object Persistence**

• **session.save(Object o)**

– Schedules insert statements to create the new

object in the database

• **session.update(Object o)**

– Schedules update statements to modify the

existing object in the database

**session.saveOrUpdate(Object o)**

– Convenience method to determine if a ‘save’ or

‘update’ is required

• **session.merge(Object o)**

– Retrieves a fresh version of the object from the

database and based on that, as well as

modifications made to the object being passed in,

schedules update statements to modify the

existing object in the database.

**Does NOT reattach the object passed in**

• **Checks to see if the object already exists in**

**the system**

– If so, gets the existing object

– Creates a new instance otherwise

• **Copies the values of the submitted object**

**into the persistent object from previous step**

• **Returns a reference to the final merged**

**object instance back to the caller**

• **Note: SUBMITTED OBJECT STATE DOES**

**NOT CHANGE**

– Merging does not affect the submitted item reference. It

is up to the developer to replace it if they so desire

**session.get(Object.class, Identifier)**

– Retrieves an object instance, or null if not found

• **session.load(Object.class, Identifier)**

– Retrieves an object instance but does NOT result in a

database call

• If managed instance not found, will return a *proxy*

– Object fully initialized when non-id attribute is accessed

» If ‘detached’, throws ObjectNotFoundException

**session.lock(Object, LockMode)**

– Reattaches a detached object to a session without

scheduling an update

– Also used to ‘lock’ records in the database

• **session.refresh(Object)**

– Gets the latest version from the database

**session.delete(Object)**

– Schedule an object for removal from the database

**Used to remove detached objects**

**directly from the database**

• **Though invisible to the user, delete() will first reattach the object via a**

**proxy, and then scheduled it for deletion – just like a normal delete**

• **session.evict(Object)**

– Removes individual instances from persistence

context, changing their state from persistent to

detached

**session.clear()**

– Removes all objects from persistence context, changing all

their states from persistent to detached

• **session.replicate(Object, ReplicationMode)**

– Used for persisting records across databases

– ReplicationModes

• EXCEPTION: throw exception if row exists

• IGNORE: don’t create if already exists

• LATEST\_VERSION: choose the latest version to save

• OVERWRITE: overwrite existing row

**Scheduled Changes**

**Session session = SessionFactory.getCurrentSession();**

**// ‘transient’ state – Hibernate is NOT aware that it exists**

**Account account = new Account();**

**// Transition to the ‘persistent’ state. Hibernate is NOW**

**// aware of the object**

**// schedules the insert statements to create the object in the database**

**session.saveOrUpdate(account);**

**// modification of the object will automatically be**

**// scheduled because the object is in the ‘persistent’ state**

**// (actually alters the initial insert statement since it hasn’t been sent yet)**

**account.setBalance(500);**

**// flushes changes to the database and commit the transaction**

**session.getTransaction().commit();**

**Differences**

1. **Save(), persist()**

**suppose if our generator class name is Increment means hibernate itself 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**

**2)update(), merge()**

Both update() and merge() methods in hibernate are used to convert the object which is in detached state into persistence state. But there is little difference.

**Update():- if you are sure that the session does not contains an already persistent instance with the same identifier then use update to save the data in hibernate**

**Merge():-if you want to save your modifications at any time with out knowing about the state of an session then use merge() in hibernate.**

**update raise the error if use record not available but merge can't instead it will create the record.**

**3)get(), load()**

The get( ) methods always hit the database. Meaning, as soon as the call to get( ) occurs, Hibernate issues an SQL statement to the database in an attempt to fetch the associated data (usually a row in the database) to rebuild the requested persistent object. A call to load( ), on the other hand, does not immediately incur a call to the database. The load( ) method causes a proxy object to be constructed as a stand-in for the persistent object. It is only after some state is requested from the proxy that Hibernate issues the appropriate SQL to the database and builds the real persistent object. **When using get( ), the method will return null if no data exists for the requested identifier.Since the load( ) method does not immediately retrieve the object, if no data exists for the identifier used to retrieve the object, an ObjectNotFoundException is thrown once data is requested of the proxy.**

Only use the load() method if you are sure that the object exists.

If you are not sure that the object exists, then use one of the get() methods

load() method will throw an exception if the unique id is not found in the database.

get() method will return null if the unique id is not found in the database.

load() just returns a proxy by default and database won’t be hit until the proxy is first invoked.

get() will hit the database immediately.

**4)evict(), clear()**

evict() deletes the data from the session only,but delete() deletes the data from both the session as well as DataBase.

The method evict() removes a single object from the session, clear() removes all objects from the session. So calling clear() is like calling evict() on every single object associated with the session.

* By default, for each hibernate application, the first level cache is automatically been enabled
* As a programmer, we no need to have any settings to enable the first level cache and also we cannot disable this first level cache
* the first level cache is associated with the session object and scope of the cache is limited to one session only
* When we load an object for the first time from the database then the object will be loaded from thedatabase and the loaded object will be stored in the cache memory maintained by that sessionobject
* If we load the same object once again, with in the same session, then the object will be loaded from the local cache memory not from the database
* If we load the same object by opening other session then again the object will loads from the database and the loaded object will be stored in the cache memory maintained by this new session
* Session ses1 = factory.openSession();
* Object ob1 = ses1.get(Student.class, new Integer(101));
* Object ob2 = ses1.get(Student.class, new Integer(101));
* Object ob3 = ses1.get(Student.class, new Integer(101));
* Object ob4 = ses1.get(Student.class, new Integer(101));
* --
* session.close();
* Session ses2 = factory.openSession();
* Object ob5 = ses2.get(Student.class, new Integer(101));

**Explanation:**

* In line number1, i have opened one session with object is ses1
* In line number2, loaded one object with id 101, now it will loads the object from the database only as its the first time, and keeps this object in the session cache
* See at line number 4,5,6 i tried to load the same object 3 times, but hear the object will be loaded from the stored cache only not from the database, as we are in the same session
* In line number 9, we close the first session, so the cache memory related this session also will be destroyed
* See line number 11, again i created one new session and loaded the same object with id 101 inline number 12, but this time hibernate will loads the object from the database

**Important**

The loaded objects will be stored in cache memory maintained by a session object and if we want toremove the objects that are stored in the cache memory,  then we need to call either evict() or clear()methods.   Actually evice() is used to remove a particular object from the cache memory and clear() is toremove all objects in the cache memory

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >Session ses = factory.openSession();  Object ob = ses1.get(Student.class, new Integer(101));  Student s = (Student)ob  System.out.println(s.getStudentId());    ses.evict(s);    Object ob1 = ses.get(Student.class, new Integer(101)); |

* Opened session at line number 1
* Loaded the object with id 101, so hibernate will lads this object from the database as this is thefirst time in the session
* At line number 4, i printed my data  bla bla..
* then in line number 6, i removed this object [ with id 101 ] from the cache memory of the session by calling evict() method
* Now in line number 8 again i tried to load the same object,  so as we are in the same session hibernate first will verify whether the object is there in the cache or not, if not loads the object from the database, but we removed the object from the cache with evict() method right, so hibernate will loads from the database
* I mean, first checks at local session then only from the database if its not available in the local cache

**Cascading**

### Cascade save / update example

In this example, if a ‘Stock’ is saved, all its referenced ‘stockDailyRecords’ should be saved into database as well.

#### 1. No save-update cascade

In [previous section](http://www.mkyong.com/hibernate/hibernate-one-to-many-relationship-example/), if you want to save the ‘Stock’ and its referenced ‘StockDailyRecord’ into database, you need to save both individually.

Stock stock =**new** Stock();

StockDailyRecord stockDailyRecords =**new** StockDailyRecord();

*//set the stock and stockDailyRecords data*

stockDailyRecords.setStock(stock);

stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);

session.save(stockDailyRecords);

Output

Hibernate:

insert into stock **(**STOCK\_CODE, STOCK\_NAME**)**

values **(**?, ?**)**

Hibernate:

insert into stock\_daily\_record

**(**STOCK\_ID, PRICE\_OPEN, PRICE\_CLOSE, PRICE\_CHANGE, VOLUME, DATE**)**

values **(**?, ?, ?, ?, ?, ?**)**

#### 2. With save-update cascade

The **cascade=”save-update”** is declared in ‘stockDailyRecords’ to enable the save-update cascade effect.

*<!-- Stock.hbm.xml -->*

**<set**name="stockDailyRecords"cascade="save-update"table="stock\_daily\_record"...**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="StockDailyRecord"**/>**

**</set>**

Stock stock =**new** Stock();

StockDailyRecord stockDailyRecords =**new** StockDailyRecord();

*//set the stock and stockDailyRecords data*

stockDailyRecords.setStock(stock);

stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);

Output

Hibernate:

insert into stock **(**STOCK\_CODE, STOCK\_NAME**)**

values **(**?, ?**)**

Hibernate:

insert into stock\_daily\_record

**(**STOCK\_ID, PRICE\_OPEN, PRICE\_CLOSE, PRICE\_CHANGE, VOLUME, DATE**)**

values **(**?, ?, ?, ?, ?, ?**)**

The code **session.save(stockDailyRecords);** is no longer required, when you save the ‘Stock’, it will “cascade” the save operation to it’s referenced ‘stockDailyRecords’ and save both into database automatically.

### Cascade delete example

In this example, if a ‘Stock’ is deleted, all its referenced ‘stockDailyRecords’ should be deleted from database as well.

#### 1. No delete cascade

You need to loop all the ‘stockDailyRecords’ and delete it one by one.

Query q = session.createQuery("from Stock where stockCode = :stockCode ");

q.setParameter("stockCode", "4715");

Stock stock =(Stock)q.list().get(0);

**for**(StockDailyRecord sdr : stock.getStockDailyRecords()){

session.delete(sdr);

}

session.delete(stock);

Output

Hibernate:

delete from stock\_daily\_record

where DAILY\_RECORD\_ID=?

Hibernate:

delete from stock

where STOCK\_ID=?

#### 2. With delete cascade

The **cascade=”delete”** is declared in ‘stockDailyRecords’ to enable the delete cascade effect. When you delete the ‘Stock’, all its reference ‘stockDailyRecords’ will be deleted automatically.

*<!-- Stock.hbm.xml -->*

**<set**name="stockDailyRecords"cascade="delete"table="stock\_daily\_record" ...**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="StockDailyRecord"**/>**

**</set>**

Query q = session.createQuery("from Stock where stockCode = :stockCode ");

q.setParameter("stockCode", "4715");

Stock stock =(Stock)q.list().get(0);

session.delete(stock);

Output

Hibernate:

delete from stock\_daily\_record

where DAILY\_RECORD\_ID=?

Hibernate:

delete from stock

where STOCK\_ID=?

### Cascade delete-orphan example

In above cascade delete option, if you delete a Stock , all its referenced ‘stockDailyRecords’ will be deleted from database as well. How about if you just want to delete two referenced ‘stockDailyRecords’ records? This is called orphan delete, see example…

#### 1. No delete-orphan cascade

You need to delete the ‘stockDailyRecords’ one by one.

StockDailyRecord sdr1 =(StockDailyRecord)session.get(StockDailyRecord.**class**,

**new**Integer(56));

StockDailyRecord sdr2 =(StockDailyRecord)session.get(StockDailyRecord.**class**,

**new**Integer(57));

session.delete(sdr1);

session.delete(sdr2);

Output

Hibernate:

delete from stock\_daily\_record

where DAILY\_RECORD\_ID=?

Hibernate:

delete from stock\_daily\_record

where DAILY\_RECORD\_ID=?

#### 2. With delete-orphan cascade

The **cascade=”delete-orphan”** is declared in ‘stockDailyRecords’ to enable the delete orphan cascade effect. When you save or update the Stock, it will remove those ‘stockDailyRecords’ which already mark as removed.

*<!-- Stock.hbm.xml -->*

**<set**name="stockDailyRecords"cascade="delete-orphan"table="stock\_daily\_record"**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="StockDailyRecord"**/>**

**</set>**

StockDailyRecord sdr1 =(StockDailyRecord)session.get(StockDailyRecord.**class**,

**new**Integer(56));

StockDailyRecord sdr2 =(StockDailyRecord)session.get(StockDailyRecord.**class**,

**new**Integer(57));

Stock stock =(Stock)session.get(Stock.**class**, **new**Integer(2));

stock.getStockDailyRecords().remove(sdr1);

stock.getStockDailyRecords().remove(sdr2);

session.saveOrUpdate(stock);

Output

Hibernate:

delete from stock\_daily\_record

where DAILY\_RECORD\_ID=?

Hibernate:

delete from stock\_daily\_record

where DAILY\_RECORD\_ID=?

In short, delete-orphan allow parent table to delete few records (delete orphan) in its child table.

**Difference between cascade and inverse**

#### 1. inverse

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

##### Example

In this example, the relationship owner is belong to stockDailyRecords (inverse=true).

*<!-- Stock.hbm.xml -->*  
**<hibernate-mapping>**  
**<class**name="com.developer.ammon.Stock"table="stock" ...**>**  
 ...  
**<set**name="stockDailyRecords"table="stock\_daily\_record"inverse="true"**>**  
**<key>**  
**<column**name="STOCK\_ID"not-null="true"**/>**  
**</key>**  
**<one-to-many**class="com.developer.ammon.StockDailyRecord"**/>**  
**</set>**  
 ...

When you save or update the stock object

session.save(stock);  
session.update(stock);

Hibernate will only insert or update the STOCK table, no update on the foreign key column.

#### 2. cascade

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

Cascade values:-

Cascade having the values…….

* none (default)
* save
* update
* save-update
* delete
* all
* all-delete-orphan

cascade="save-update"

cascade="delete"

cascade="delete-orphan"

cascade="save-update, delete"

@OneToMany(mappedBy ="stock")

@Cascade({CascadeType.SAVE\_UPDATE, CascadeType.DELETE})

**public** Set<StockDailyRecord> getStockDailyRecords(){

**returnthis**.stockDailyRecords;

}

**Without cascade you need to save manually**

Stock stock =**new** Stock();

StockDailyRecord stockDailyRecords =**new** StockDailyRecord();

*//set the stock and stockDailyRecords data*

stockDailyRecords.setStock(stock);

stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);

session.save(stockDailyRecords);

##### Example

In this example, the cascade=”save-update” is declare on stockDailyRecords.

*<!-- Stock.hbm.xml -->*  
**<hibernate-mapping>**  
**<class**name="com.developer.ammon.Stock"table="stock" ...**>**  
 ...  
**<set**name="stockDailyRecords"table="stock\_daily\_record"  
cascade="save-update"inverse="true"**>**  
**<key>**  
**<column**name="STOCK\_ID"not-null="true"**/>**  
**</key>**  
**<one-to-many**class="com.developer.ammon.StockDailyRecord"**/>**  
**</set>**  
 ...

When you save or update the stock object

session.save(stock);  
session.update(stock);

It will inserted or updated the record into STOCK table and call another insert or update statement (cascade=”save-update”) on StockDailyRecord.

### Conclusion

In short, the “inverse” is decide which side will update the foreign key, while “cascade” is decide what’s the follow by operation should execute. Both are look quite similar in relationship, but it’s totally two different things.

**inverse = “true” example and explanation**

The “inverse” keyword is always declared in **one-to-many** and **many-to-many** relationship (many-to-one doesn’t has inverse keyword), **It defines which side is responsible to take care the relationship**

 Inverse = false (default) – will execute to update the relationship.

 Inverse = true – Do nothing

**Difference between save and saveorupdate?**

save() - save method stores an object into the database. That means it insert an entry if the identifier doesn't exist, else it will throw error. If the primary key already present in the table, it cannot be inserted.

update() - update method in the hibernate is used for updating the object using identifier. If the identifier is missing or doesn't exist, it will throw exception.

saveOrUpdate - This method calls save() or update() based on the operation. If the identifier exists, it will call update method else the save method will be called.

**Q: In which scenario Hibernate shouldn't be used?**

A: When application is using procedures then using Hibernate is not recommended.

**Q: What is cascading and inverse?**

"inverse" is decide which side will update the foreign key, while "cascade" is decide what’s the follow by operation should execute.

**Q: How to invoke stored procedure in hibernate?**

A: { ? = call selectAllEmployees() }

**Hibernate : Session Lock**

**Hibernate : Session Lock :-**

The lock() method obtains the specified lock level upon the given object. It allows your application   
to reattach the object to the session. It doesn't check or update the database as it assumes that the  
database is in sync with the detached object.

**Syntax:**   lock(Object entity,LockMode lockMode)  
**Specified by:**  lock in interface HibernateOperations  
**Parameters:**  
entity - the persistent instance to lock  
lockMode - the lock mode to obtain  
**Throws:**  
DataAccessException - in case of Hibernate errors

There are five kind of locks that can be obtained -

**LockMode.WRITE :** It is obtained when Hibernate updates an object or saves new object.

**LockMode.UPGRADE :**Hibernate obtains this lock when using the SELECT <String> FOR UPDATE SQL command.

**LockMode.UPGRADE\_NOWAIT :** Hibernate obtains this lock when using the SELECT <String> FOR UPDATE SQL command under the Oracle database server.

**LockMode.READ :** Hibernate obtains this lock either at the user's request or when neede for reading an object.

**LockMode.NONE :** Hibernate obtains this lock when a transaction finishes or during the start of a call to update() or saveOrUpdate().

**Example :**

### Which type of locking mechanism use in hibernate ?

We can implement two types of locking mechanism in Hibernate as follows  
  
  
1. **Optimistic Lock**:---  
                             when transaction acquires  the**Optimistic lock** on a column or row then other transaction can read the locked column or row data but can't update the data.  
  
  
2.  **Pessimistic Lock**:---  
                           when transaction acquires the **Pessimistic lock** on a column or row then other transaction can't read or update data.  
  
  
Hibernate  prefer optimistic locking for better concurrency.

Default value is true.  
when Hibernate decide to lock then optimistic make false.  
<optimistic-lock="false">  
  
  
Possible values are:

* **none**
* **version**
* **dirty**
* **all**

Default value is **version**  
  
**<class name="" table="" optimistic-lock="version">**  
  
  
**-----**  
**<set name=""  optimistic-lock="false">**  
**----**  
  
**</set>**  
  
**</class>**

==========================

**What is optimistic locking?**

Optimistic locking is a method used to prevent simultaneously access problem on the same entity for change, which does not lock on the database level, in order to maintain the correct data.

For example, if two user wants to update same entity in different transactions, when first one saves and then second one saves without getting updates from the first user’s change, who will be the winner?

Most of the web applications, like ours, has such cases: Two users retrieve and modify a data on the page, then first user saves the data a transcation and then second user modifies and saves the data in an another transacion. Here are 3 alternatives:  
1. Last commit wins : Both updates are performed, the second user overwrites without seeing the first user changes and without any error message  
2. First commit wins : (Optimistic Locking) The update is performed, but second user gets error message, requiring restart of the processes on the first users changes.

3. Merge conflicting update : The second user are prompted to merge changes.

**Automatic Versioning**

Hibernate provides automatic versioning for options 2 and 3, using a version field managed by hibernate.

**Versioning with integer version**  
  
public class MyClass {  
...  
private int version;  
...  
}

<class name="MyClass">  
<id ...>  
<version name="version" column="VERSION" access="field">  
...  
</class>

**Versioning with timestamp**  
  
public class MyClass {  
...  
private int lastModifyDataTime;  
...  
}

<class name="MyClass">  
<id ...>  
<timestamp name="lastModifyDataTime" column="LAST\_MODIFIY\_DATE\_TIME" access="field">  
...  
</class>

Automatic versioning is handled by hibernate, you don’t need to update versions/timestamps.  
Considering the example mentioned above, when second user saves, hibernate finds this user is working on the stale data and throws **StaleObjectException**.

You can catch this exception, and rethrow your own exception. By the way, if you’re using Hibernate with Spring, this exeception is wrapped with **HibernateOptimisticLockingFailureException**.

One more point, you should be aware of that hibernate ignores the versioning when getting object and updating fields and then version are in the same session.

==============================================

## Optimistic vs. Pessimistic Locking

[**Optimistic locking**](http://www.martinfowler.com/eaaCatalog/optimisticOfflineLock.html) takes the “optimistic” view that data corruptions due to concurrent edits will occur rarely and no locking is needed, so it’s more important to allow concurrent access than to lock out concurrent updates. If a conflict occurs, the transaction must be aborted and repeated. Frameworks which support optimistic locking typically maintain a version field and raise an exception if one tries to update an object with an outdated version number.

[**Pessimistic locking**](http://www.martinfowler.com/eaaCatalog/pessimisticOfflineLock.html) takes the “pessimistic” view that users are highly likely to corrupt each other’s data, and that the only safe option is to lock the database and serialize data access, so at most one user has control of any piece of data at one time. This ensures data integrity, but reduces speed and the amount of concurrent activity the system can support.

[**Stackoverflow.com**](http://stackoverflow.com/) has an entry about [**optimistic vs. pessimistic locking**](http://stackoverflow.com/questions/129329/optimistic-vs-pessimistic-locking) – when to use optimistic and when to use pessimistic locking. It says correctly that the optimistic strategy is most useful for web applications where you do not necessarily maintain a connection to the database for your session. In this situation the client cannot actually maintain database locks as the connections are taken from a pool and you may not be using the same connection from one access to the next.

Maybe the following analogy is helpful to understand the difference between both kinds of locking, too, although it is a bit slippery:

**Optimistic locking** is like leaving the door to the toilet open: you have a number of toilets, select one, and you are optimistic that nobody will come to use the same toilet, too. If someome comes, it will of course be embarassing, an exception is raised, and he must abort his try, but you hope that these conflicts are the execption.

**Pessimistic locking** is like always locking the door to the toilet: although there are a number of different toilets, you are expecting every time that somebody else will come to use one, too, and that the only safe option is to lock all toilet doors (which corresponds to page locking). If someone comes and wants to use a toilet, he cannot enter and must wait before the locked door.

In Ruby on Rails you can do both. [**Pessimistic Locking**](http://api.rubyonrails.org/classes/ActiveRecord/Locking/Pessimistic.html)is possible through row-level locking using SELECT … FOR UPDATE and other lock types. [**Optimistic Locking**](http://api.rubyonrails.org/classes/ActiveRecord/Locking/Optimistic.html) means to hope for the best and handle the ActiveRecord::StaleObjectError.