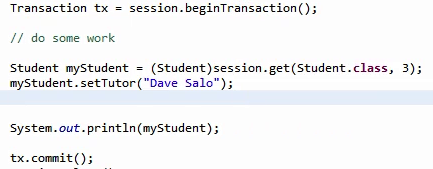
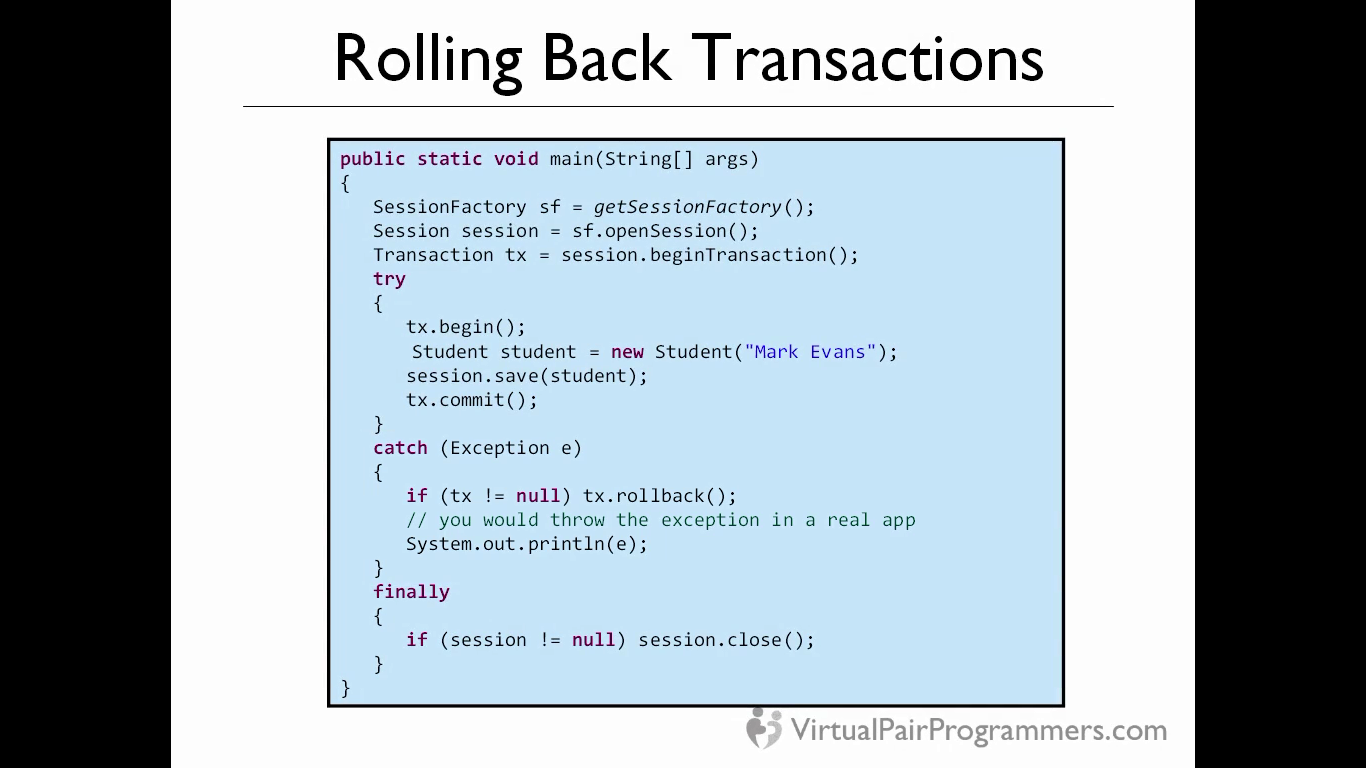
-To delete an object in Hibernate – find an object first and then the object will be persisted. Then use delete function to delete. Alternatively we can use **query** language to delete in 1 statement.



-While updating an entity, we don’t need to specify any operation on session object. Because whenever we are committing a transaction, Hibernate will check for all the objects (**automatic** **dirty check**) that were modified and issue and internal update for those objects.

*This is a big advantage of hibernate/ORM language that it intelligently checks if the object is updated or not. This is straight forward in above example but for a complex logic, we cannot be sure every time that an object is changed or not and in SQL, we may issue an unnecessary update statement*.

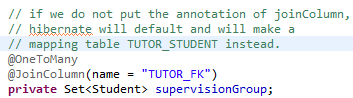
Important – When we declare a field as int(primitive), hibernate will add a not null constraints over the field. We can also use a wrapper Integer to have null values for the field.



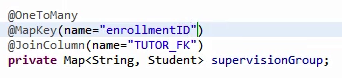
**OneToMany annotations(unidirectional)**

<http://stackoverflow.com/questions/11938253/jpa-joincolumn-vs-mappedby>

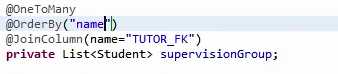
@joinColumn is used in two ways for unidirectional mapping. When used in “Many “side, this is used to change the column name. When used in “One” side, this mapping will force the foreign constraint on the “Many” side of the relationship instead of the link table. The below code is only for unidirectional relationship as the Student entity has no link for Tutor class below.



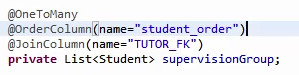
A oneToMany can be mapped by using a Map collection also. Here we will have to define a @MapKey annotation to tell hibernate that while building a map, we will use enrollmentID from Student entity as a key of the map. So when finding a tutor, hibernate will build a map.



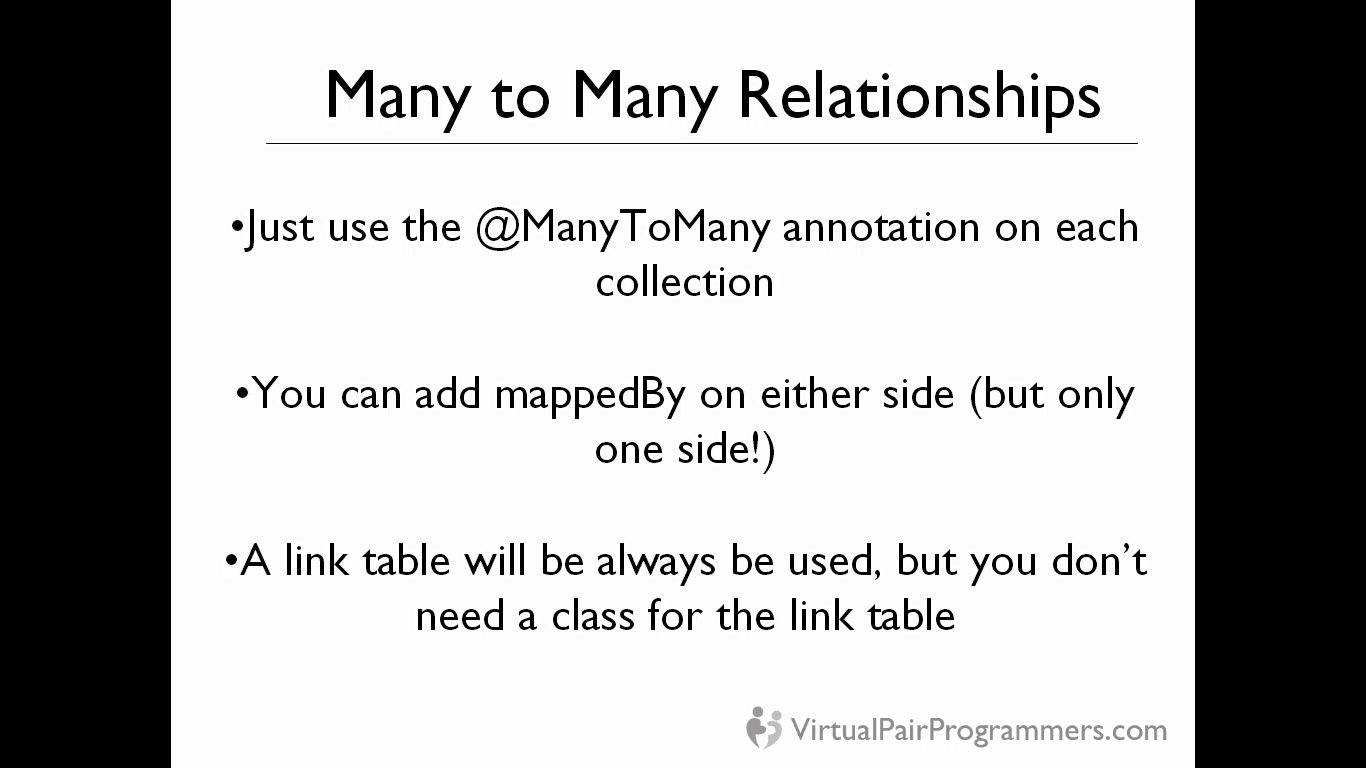
We can also use a List interface which will identify the order of inserts. Additionally we have an @OrderBy annotation to work with.

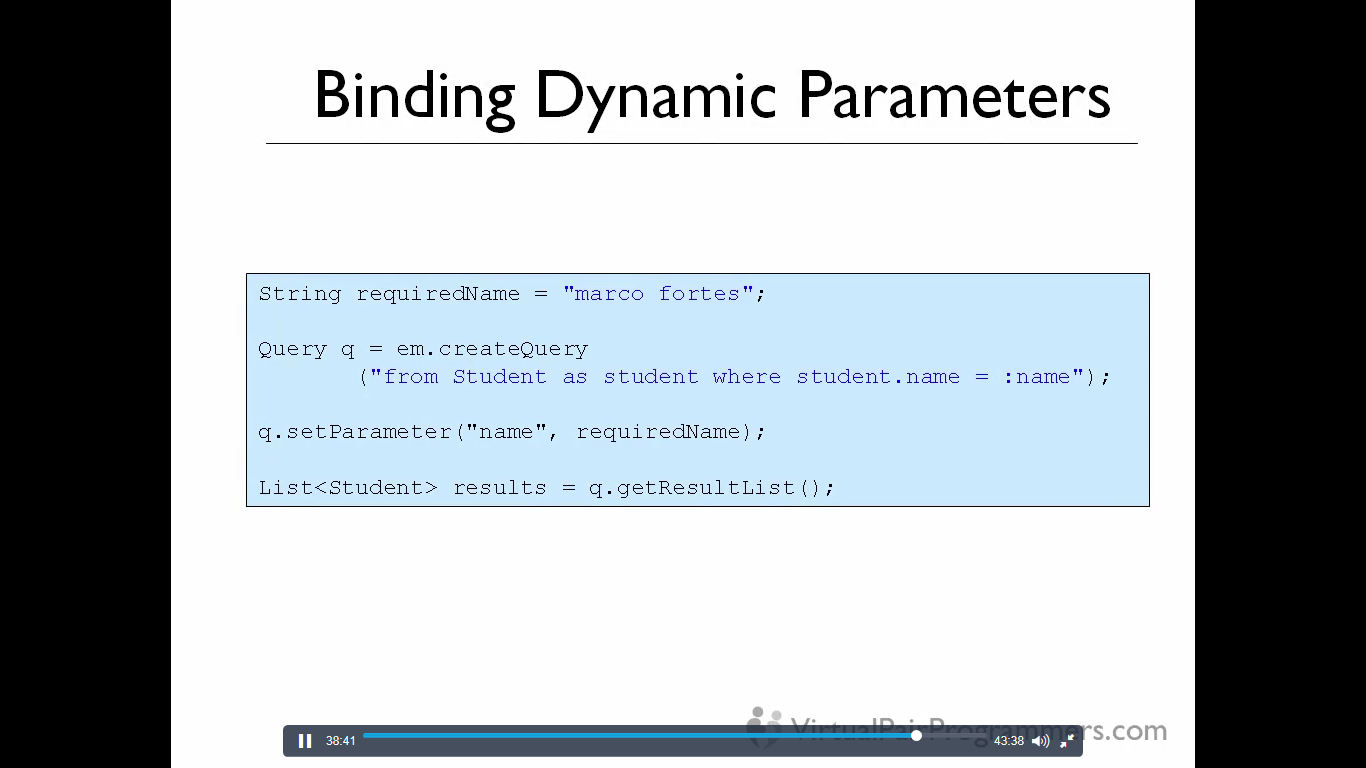


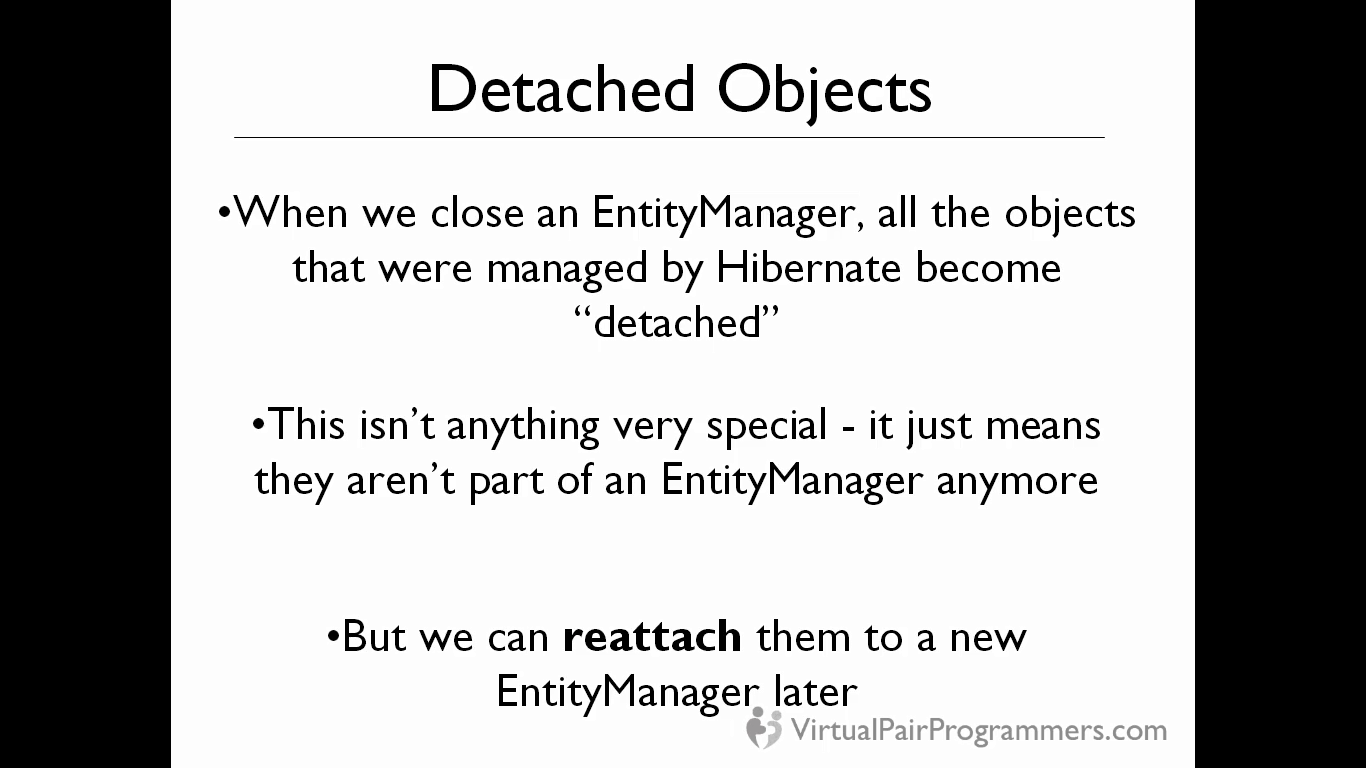
We can also provide an additional column to tract the order by using @orderColumn



**OneToMany annotations (Bidirectional)**



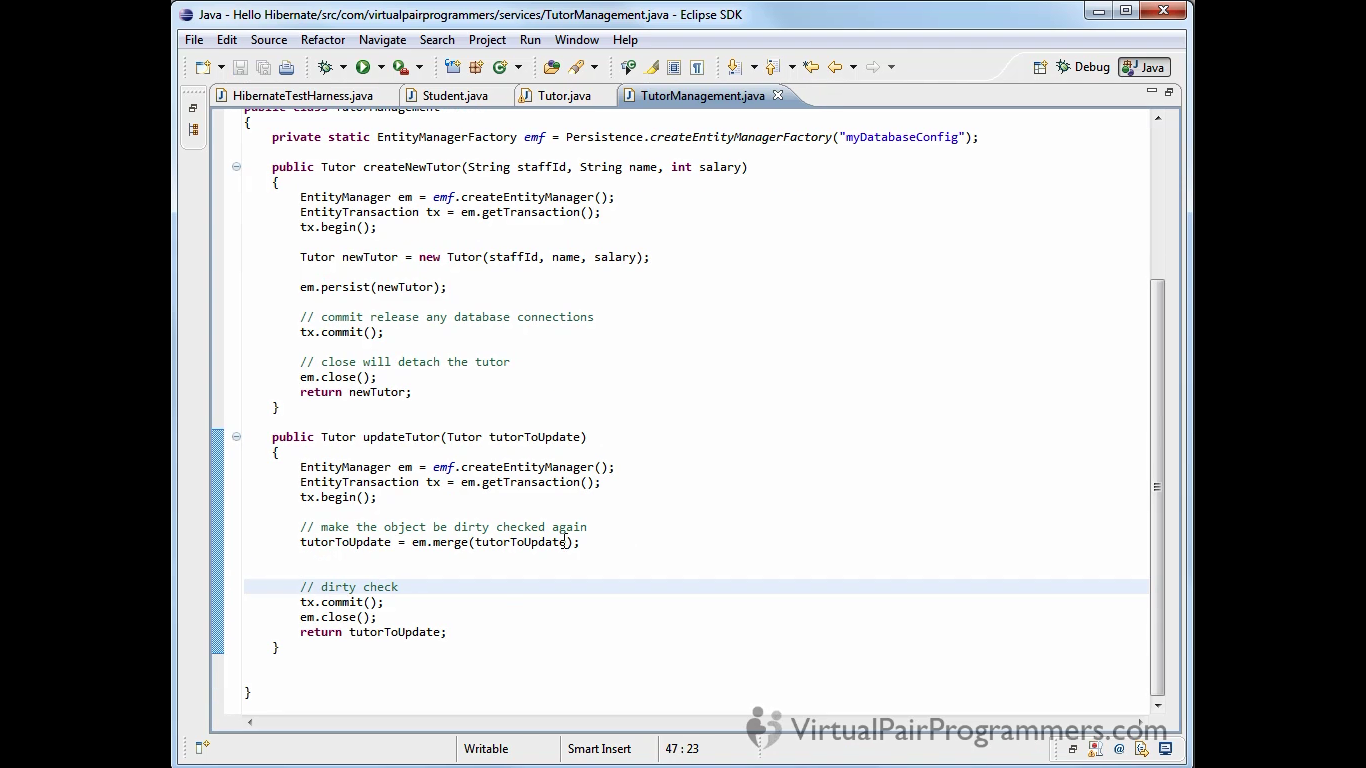




When we have a process which is long waiting process, we should close our entity manager as that is not thread safe.

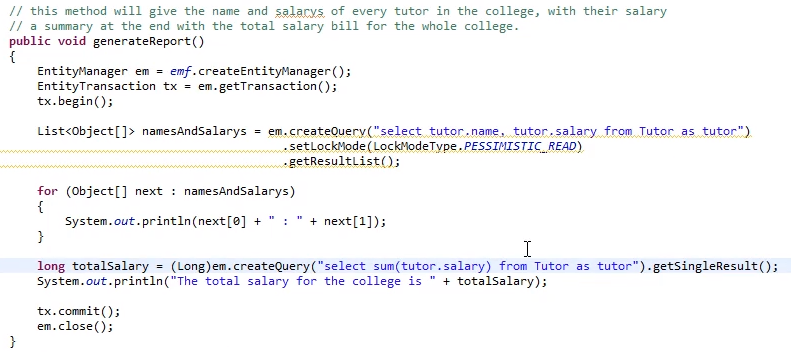
But once we close the EM, all the objects that were held by the EM, will be released and any updates on those objects will not result in dirty checking and hence will not be reflected in DB.

So we can use a detached object concept, then when we are finished with the work, we can again reattach and detached object, then it will be picked by ORM and all the updates will be executed.



**Pessimistic Locking** –

There are two types of pessimistic locks – Pessimistic Read/Pessimistic Write.



From above example, if we do not use setLockMode in the query, then before coming and executing the final totalSalary query, the data have potential for corruption. As some different transaction may happen to come and change the values for salary and our sum total will be different. So the two queries will not be in sync if we haven’t used the pessimistic lock.

When we use this lock, it will lock the rows for the read purpose and any parallel transactions will have to wait until the commit () line to get the lock on rows.

Pessimistic\_Read = we want a consistent data for read during our transaction and will prevent any other transaction for update. But as we are not updating any rows in the above example and demanded only Read lock, other transactions are free to acquire the read lock on the table but not write lock.

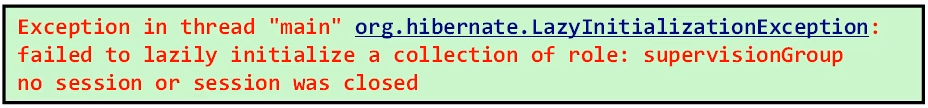
But this read lock has disadvantage as this can lead to deadlock if we do updates in our read only transactions. This is unrealistic but can lead to deadlock if not properly used.

**Performance Tuning**

**Lazy Initialization** –

It works with a concept of **proxy** object which is a temporary object without all the list of collection a parent Tutor object can hold. So this concept is used just in case if we do not use the collection of Students object. So proxy is smaller object than the original Tutor object. But it is clever as when we need a collection, it will automatically fetch the list. So this way Hibernate will avoid loading collection if they are never used.

**Problem** – Lazy collection fetching only works inside the scope of an EntityManager. If we access the collection outside the session it will give below error.

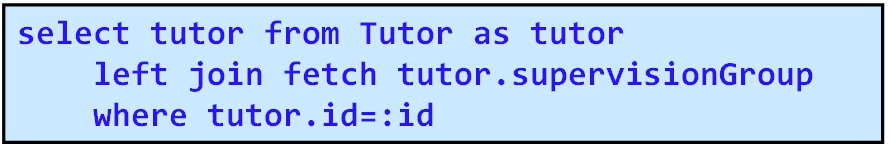


So we can switch off the default lazy initialization process by following the below annotation



This technique is not used very often as it will load the entire collection object even in cases where we would want only Tutor object and not Student.

Instead we can use a local eager fetch strategy in those queries only where we want the collection object and not on the collection declaration as annotation.

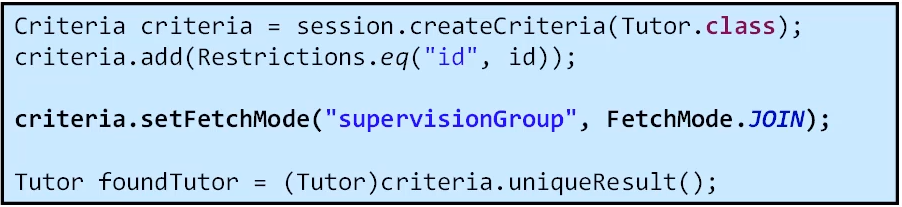


**Important** – If we omit the word left in the query, hibernate will issue an inner join between Tutor and Student.

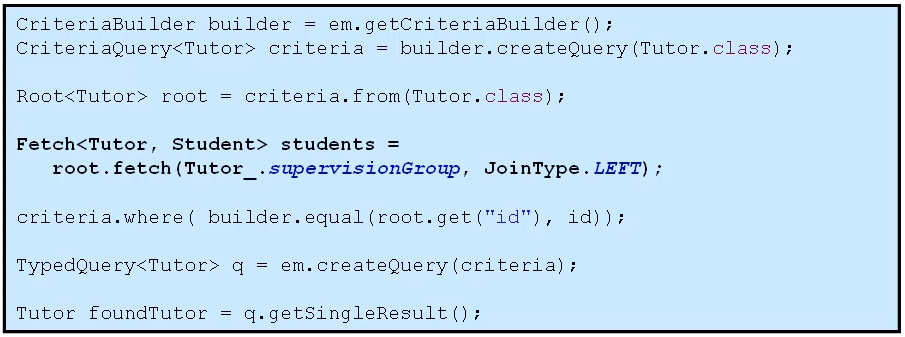
Left join fetch = Left outer join – will return the entire Tutor even if the supervisionGroup for that tutor is empty. Even if the parent object does not have a child object, it will be returned in the query. And this is what should work. Say – find all tutors irrespective of students.

Join fetch = inner join – will return only those Tutors who have students collections in supervisionGroup. Say – find all tutors who have students.

Below is the Criteria way of fetching the list of Tutors with Students that is Eager fetch. FetchMode.Eager is deprecated so always use FetchMode.JOIN.



JPA way of fetching the same thing



We can also solve this problem by using below methods for web application

* Open session in view
* Or Open Entity Manager in view.

**Important** – Single valued relationship defined by annotations are EAGER by default. Example – Student has one tutor, so fetching student will default fetch TUTOR.

So to illustrate it with example, fetching the list of all students is a very simple query but hibernate will issue 3 selects statements for fetching the students as it will also fetch associated Tutor objects eagerly from the DB.

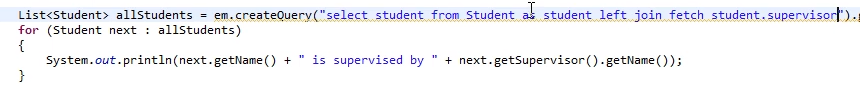
One select for list of students – another for student 1 tutor, student 2 has same tutor as student1 so no select. And a 3rd select for student 3 to fetch associated new tutor into memory.

This is called (***N+1 Selects***) problem – one select for parent and one select for each child object. But this is case only when we are working with **annotations but not in xml configurations**.

It is very simple to override default when working with single values entities like below in Student entity. So that will maintain consistency with collections.



But n+1 select problem can come again if we issue a get command to get the tutor name also which is described in below example SOP statement.



So, to overcome the problem, we can again use a left outer join in the query and only one query will be fired in DB. But what if there are too many records, the above query will result in one giant query and is again a performance problem.

**Batch Fetching** –

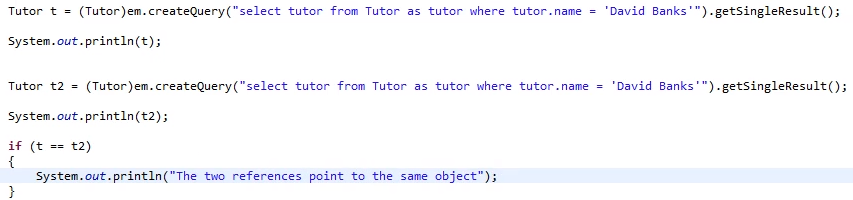
This will eliminate small n+1 selects and also a one giant select in above problem.

We provide annotation value over the Entity to specify how many entities will be queried with the Student entity. Note- this won’t work for the collection as we cannot restrict on how many numbers from list of collection should be fetched in oneToMany associations.

**First Level Cache**

Firstlevel cache works with the IDs only. It means then when we issue get or find by ID, no select command will be executed and a cached object will be returned. But look at the below example,

Here hibernate has no idea and it cannot run a query to check in EntityManger cache to find if David Banks exists or not as first level cache works only with ID. So when we executed the below command, two selects will be issued, but after the select, hibernate will recognize the returned object ID and only the cached object will be returned. The result of the second query is discarded. It will inspect the ID of the returned object and will match with the cached object ID.

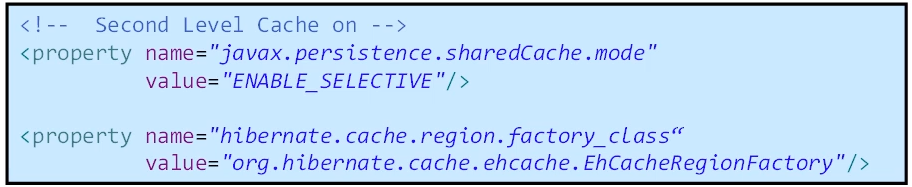
****

**Second Level Cache**

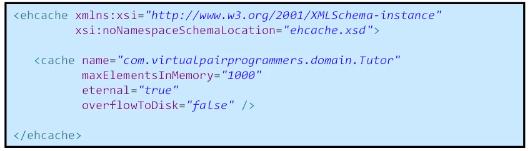
Firstlevel cache works with EntityManger only and when the transaction completes, this cache is destroyed. But the 2LC will store the objects in RAM and even after the transaction and session completes, the data would be there in memory. It would be for entire application.

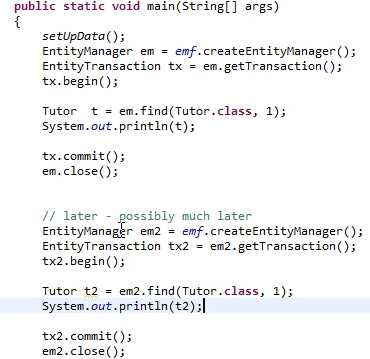
We will use the most common cache implementation and that is **EhCache.**  This will work only in single Vm and not for the distributed applications.

To enable Second level cache (2LC) –



This will enable only selective caching and not all the entities will be cached. We will have to selectively put annotations over entities to make it cacheable. And also configure the cache properties in the ehcache.xml





In the above example, em is issuing the find for tutor with id = 1. We have put the Tutor in the 2LC so even when the em is closed, the object will stay in the memory. Please note – when em is closed, the 1LC is removed.

Later when the em2 is initialized, the find on tutor with id = 1 will not issue any select statement and Tutor object will be picked from 2LC.

So to enable 2LC follow below steps

1. Configure the xml for the correct implementation
2. Annotate the entity that is to be cached
3. Configure the ehcache.xml

To test if there is some performance improvement and to get information if the 2LC is working we can collect the statistics which is only in hibernate and not in JPA.



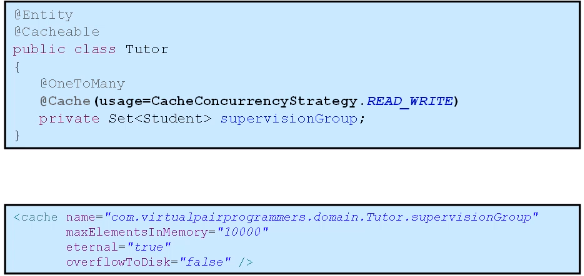


Also for classic hibernate to enable cache on separate entity is



**Caching an Associations**

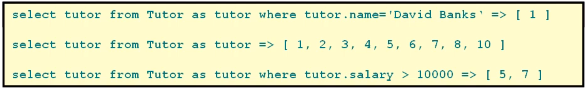
By default- 2LC will not put all the associations for Tutor class. IF we put the @Cache annotations over the Student entity, it will put the objects in the cache but will not associate with the Tutor. So when we want to retrieve all the Students for the Tutor, a select statement will be issued.



Surprisingly, the associations also need to be configured in the ehcache.xml with the fully qualified name.

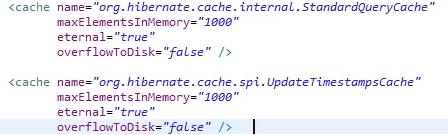
**Query Cache**

Second level cache also works with only IDs and it will not cache result for the sql statement.

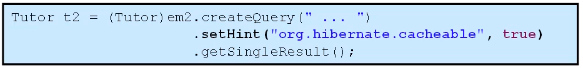


Hibernate will store each configured query with the result IDs.

To enable query cache –







1. Enable two cache regions in ehcache.xml
2. Enable query\_cache is persistence xml
3. Use setHint function to query specific use

Use the query cache for the tables which are very rarely updated and with no so many queries on.

Hibernate only properties which are not available in JPA

1. sessionFactory stats to get the hits in cache
2. cache concurrency is only available in hibernate
   1. Read\_only – if we update object in this mode, we will get error. Only suitable if the objects are never been updated.
   2. Nonstrict\_read\_only – they are not locked in cache, we might see some old data but is good in performance.
   3. Read\_write – safest, object will be locked while updating
   4. Transaction – only suitable for distributed application.
3. Caching an associations