**What is Hibernate.**Hibernate is an object-relational mapping (ORM) tool that maps POJO - plain old Java objects - to relational database tables.Hibernate is implementation of JPA.

**How hibernate maps POJO and database tables.**There are 2 ways of mapping POJO & database tables.  
 a. JPA annotations (Preferred)  
 b. XML Configurations.

**Advantages of Hibernate over jdbc?**

a. **Bolier Plate Code**: Hibernate eliminates all the boiler-plate code that comes with JDBC.

b. It supports HQL.

c. **Transaction Management**: In JDBC transactions have to managed by ourselves whereas Hibernate manages transactions implicitly.

d. **Exception handling** : JDBC throws a *checked exception (SQLException)* called thereby mandating the developer to write try-catch blocks for exception handling

Hibernate throws *unchecked exceptions* (HibernateException). This helps developers to avoid writing multiple try-catch blocks to handle exceptions.

e. Hibernate Supports **Caching**.

**Important Interfaces Of Hibernate.**

a. Session  
b. SessionFactory  
c. Transaction

**What is Session**

A session is an object that maintains the physical connection between Java application and DB.   
Session also has methods for performing CRUD operations (from DB) using methods like *persist(), load(), get(), update(), delete(),* etc.   
Additionally, it has factory methods to return Query, Criteria and Transaction objects.

**What is SessionFactory**

SessionFactory provides an instance of Session.   
It is a class that gives the Session objects based on the configuration parameters in order to establish the connection to the database.  
The application generally has a single instance of SessionFactory.

The internal state of a SessionFactory which includes metadata about ORM is immutable, i.e once the instance is created, it cannot be changed.

**Important Annotations**

@Entity is used to mark a Java class as an entity, which means instances of this class represent persistent objects that need to be stored in a database.   
It Maps Java class to the table

@Table is used to customize the database table settings associated with an entity.  
When we use @Entity without specifying @Table, Hibernate will create a table with a name that matches the class name.  
Using both annotations together allows us to define both the entity and the table properties.

@Id - used to define primary key  
@Column - used to define column name.

EmbeddedId, Generated Value, Transient,etc.

**Mapping**

One-to-One : use @OneToOne annotation.  
Many-to-One  
Many-to-Many : use @ManyToMany annotation.

**What is hibernate configuration file. Is it different from application.properties**

a. It contains database specific configuration, which is used to initilize sessionFactory.  
b. Conventionally, it should be named as hibernate.cfg.xml. It is placed under src/main/resource folder.  
c. List of Properties used in cfg file   
 i. hibernate.dialect  
 ii. hibernate.connection.driver\_class  
 iii. hibernate.connection.url  
 iv. hibernate.connection.username  
 v. hibernate.connection.password  
 vi. hibernate.connection.autocommit

Hibernate.cfg.xml is different from application.properties

**What is hibernate mapping file.**

This mapping file instructs Hibernate — how to map the defined class or classes to the database tables.  
Instead of using xml file we can use annotations as well.  
Conventionally it is named as <className>.hbm.xml.  
 a. <hibernate-mapping></hibernate-mapping> -> it is root element.  
 b. class -> specifies the persistent class. persistent class is POJO file  
 c. id -> specifies the primary key attribute to the class.  
 d. generator -> used to generate primary key.  
 e. property -> specify property name of persistent class.

eg.

<hibernate-mapping>  
 <class name = "com.Employee" table = "empTable">  
 <id name = "id"> <generator class = "assigned"></generator> </id>  
 <property name = "firstName"></property>  
 <property name = "lastName"></property>  
 </class>  
</hibernate-mapping>

**Steps to Create Sample App for Hibernate.**

a. Create Persistent Class(POJO)  
b. Create the mapping.  
c. Create the configuration file.  
d. Create class for creating and storing the persistent POJO.  
e. Run the application to see results.

**Difference between openSession() & getCurrentSession()?**

openSession()   
 helps in opening new session.  
 We should close the session object once we are done with DB operations.  
 In a multi threaded environment we should open a new session for each request.

getCurrentSession()

returns the session bound to the context.  
 Since this session object belongs to context of hibernate, it is okay if we don't close it.

**Some Methods in JPA Repository interface.**findAll(), findById(), getById(), saveAll(), findAllById()

**Fetch Type**

1. Eager (whole data loads at once)
2. Lazy (data loads only when we call getter or size method)

By default, lazy loading is implemented.  
e.g. @OneToMany(mappedBy= “question”, fetchType=FetchType.Eager)

**Difference between Session get() & load() method?**

|  |  |
| --- | --- |
| Get() | Load() |
| return null if object is not found in cache as well as on DB. | throws ObjectNotFoundException if object is not found in cache as well as on DB, but never returns null. |
| involves in DB hit if object does not exist in session cache and return a fully initialized object which may involve several DB call. | returns proxy Object and only initialize the object or hit the DB if any method other then getId() is called on persistent or entity Object. This is lazy initialization which increases performance. |
| should be used when we are not sure that Object exist in DB or not. | should be used when we are sure that Object exist in DB. |

**Why Hibernate Caching and types?**

Why Caching? -> Makes application faster and improves performance. The idea behind caching is to reduce DB queries.

1. First Level Caching
2. Second Level Caching
3. Query Level Caching

**First Level Caching.**

The first-level cache is the Session cache and is a mandatory cache through which all requests must pass.

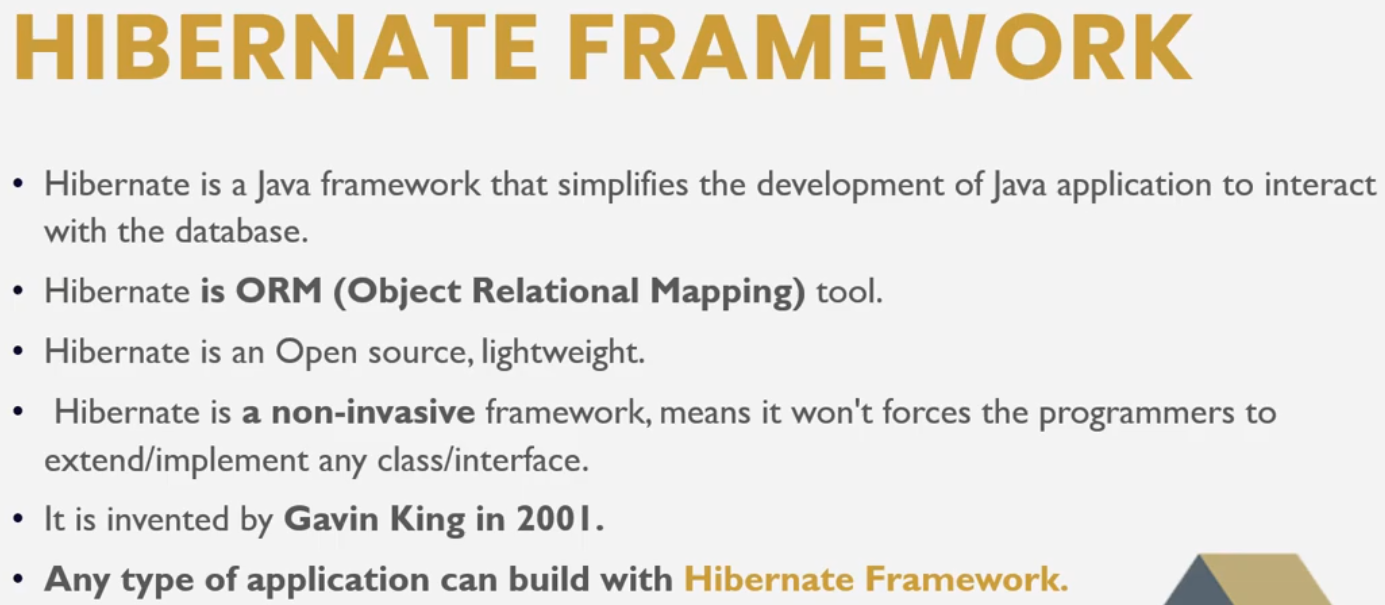
It is by default enabled and there is no way we can disable it. But Hibernate provides methods so that the cache can be cleared.

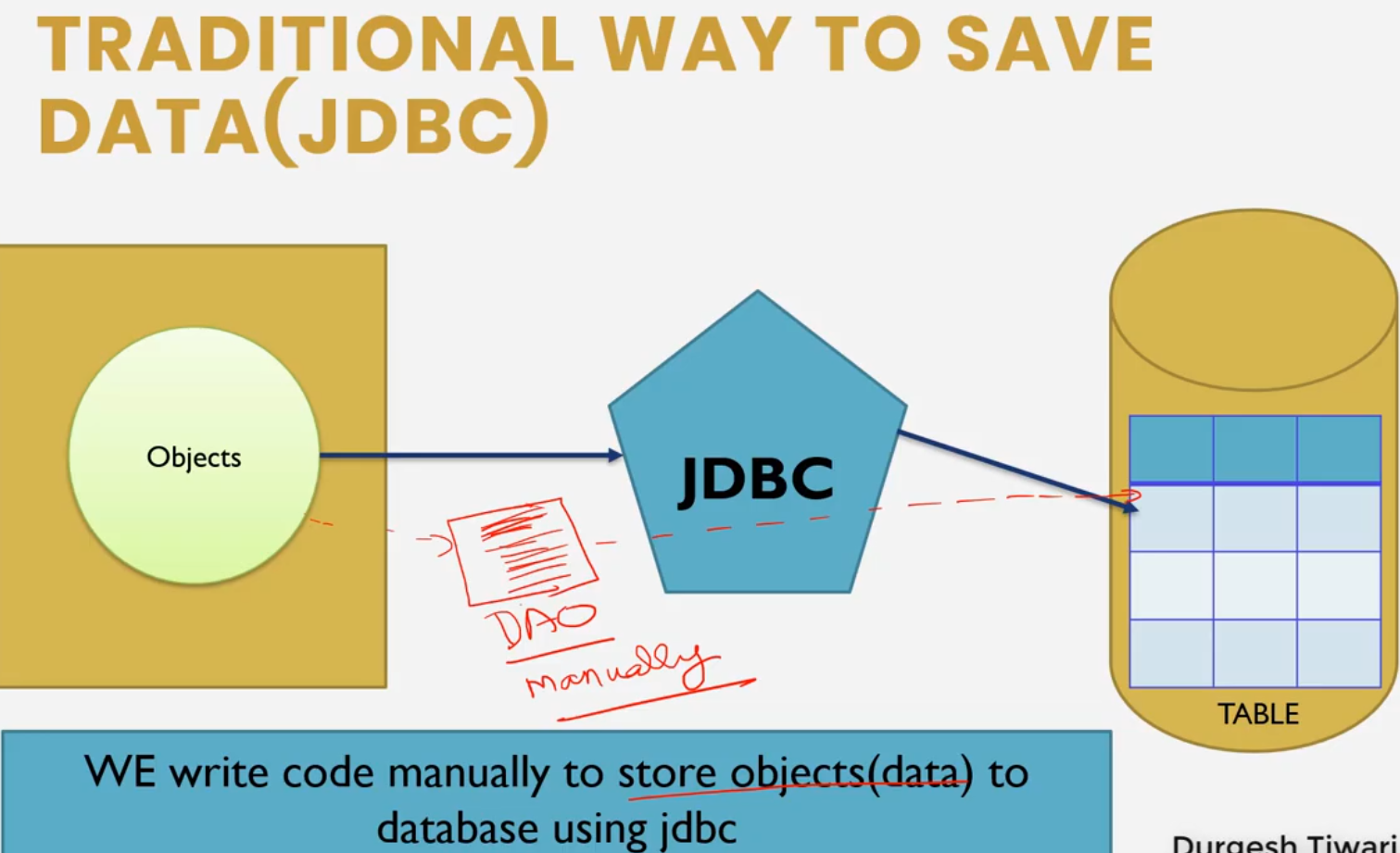
\* An object cached in a session will not be visible to other sessions and when session is closed; all the cached objects will also be lost.

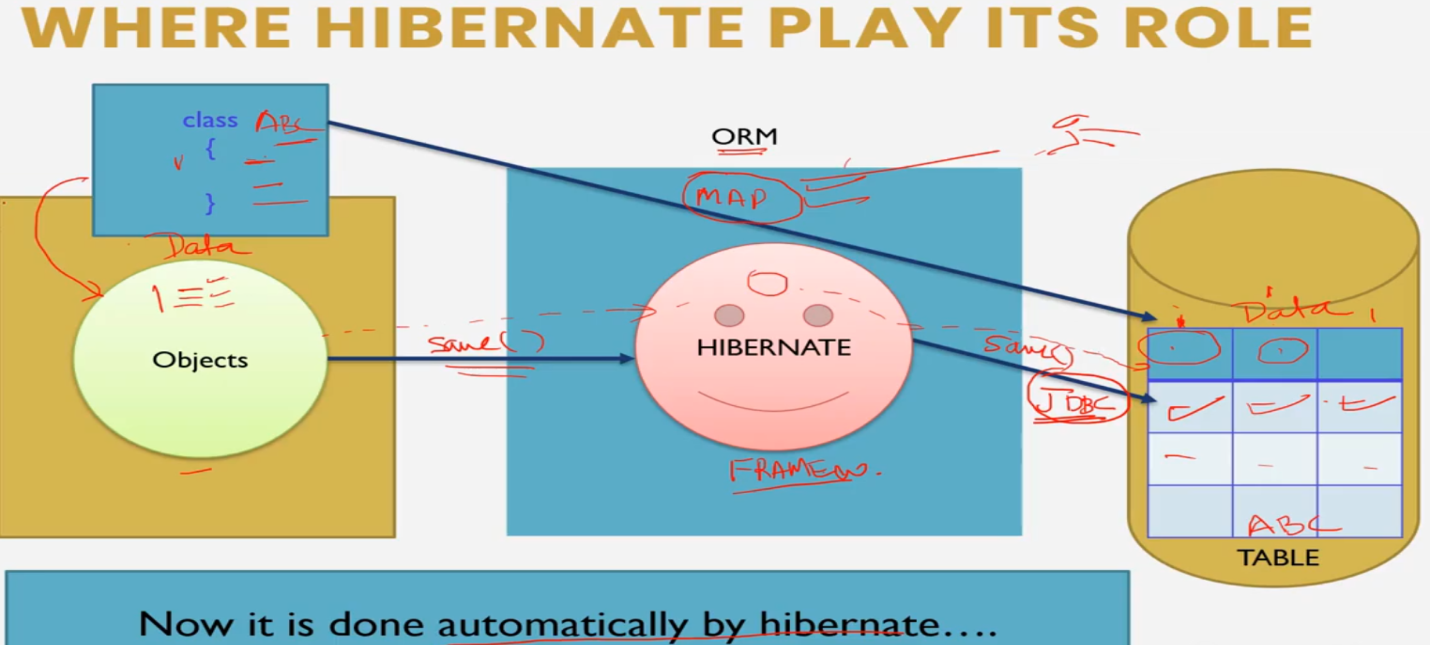
**Second Level Caching.**

The second-level-cache is optional and disabled by default. We can enable it by configurations.  
Currently EHCache and infinispan provides implementation for Hibernate Second Level Cache.

**Query Level Caching.**Query Level Cache should be used in conjuction with Second Level Cache.It never works with first level cache.





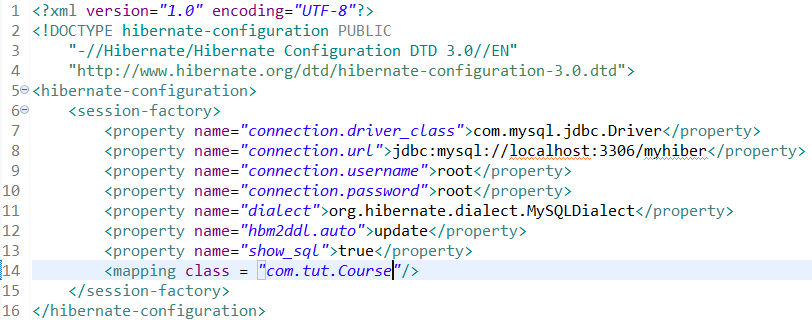


* Add Hibernate Core and Mysql Connector dependency in POM.xml

**Adding hibernate Configuration**

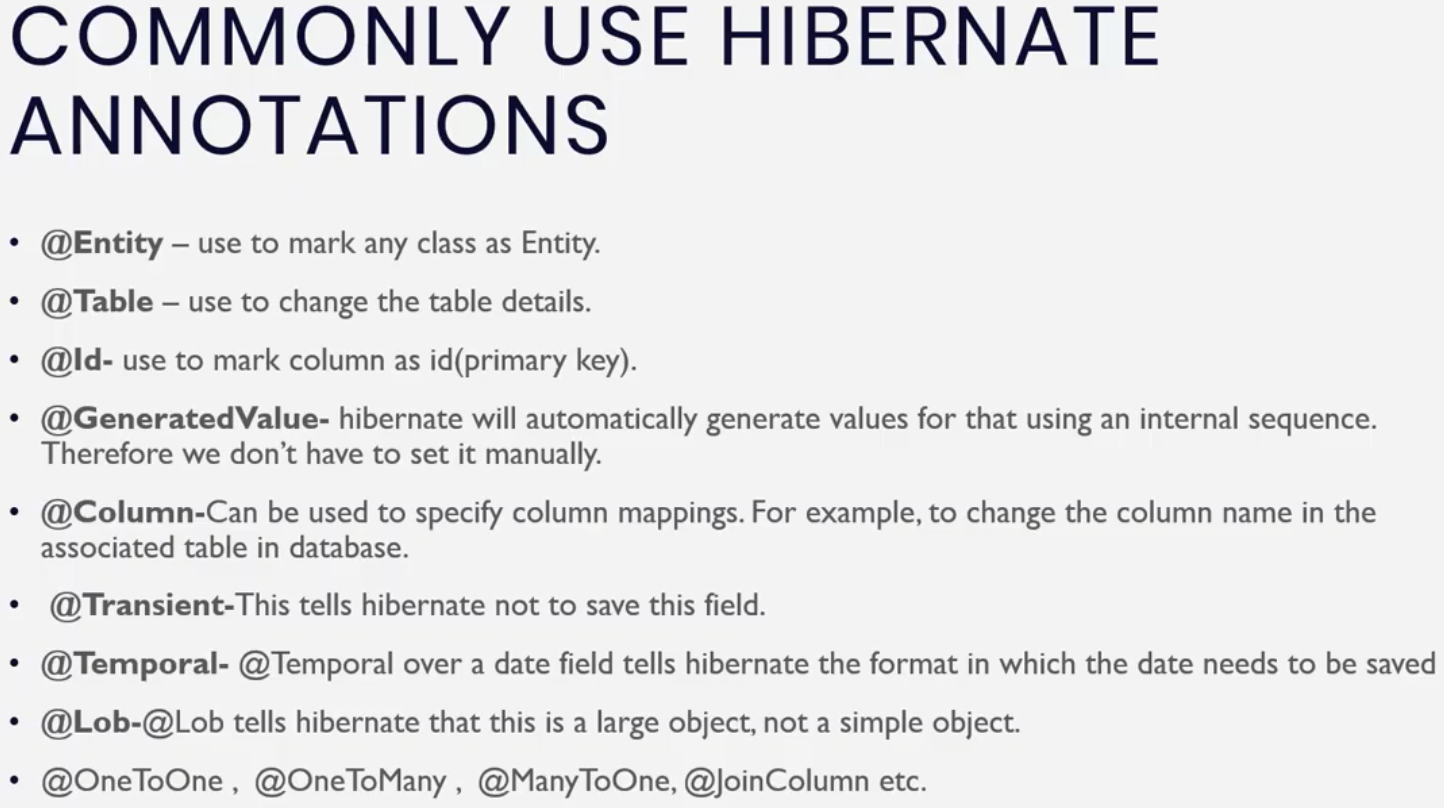
Right click on src/main/java add a xml fle hibernate.cfg.xml

Add the following dtd configuration file

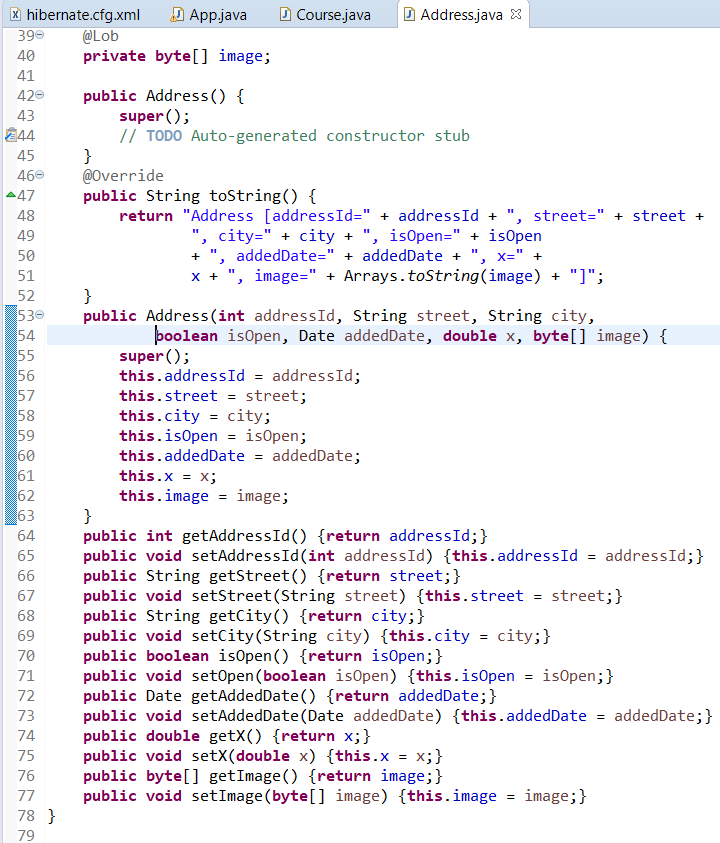
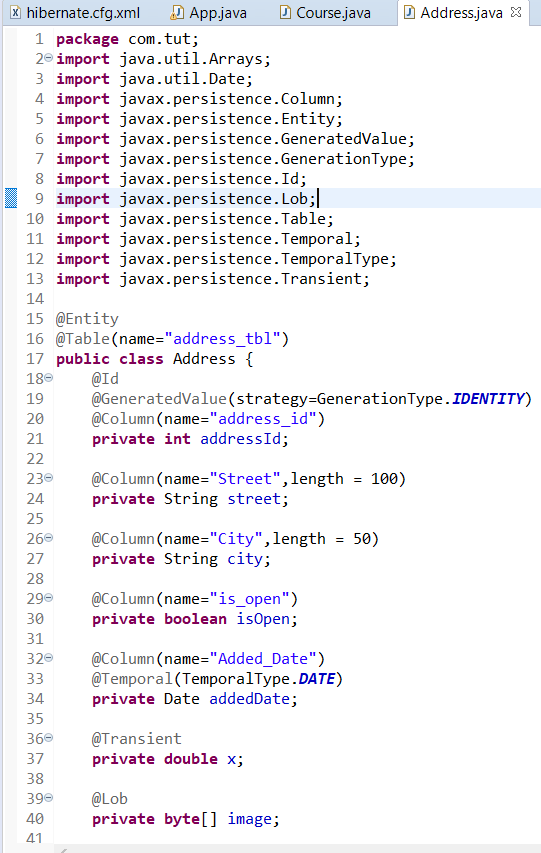
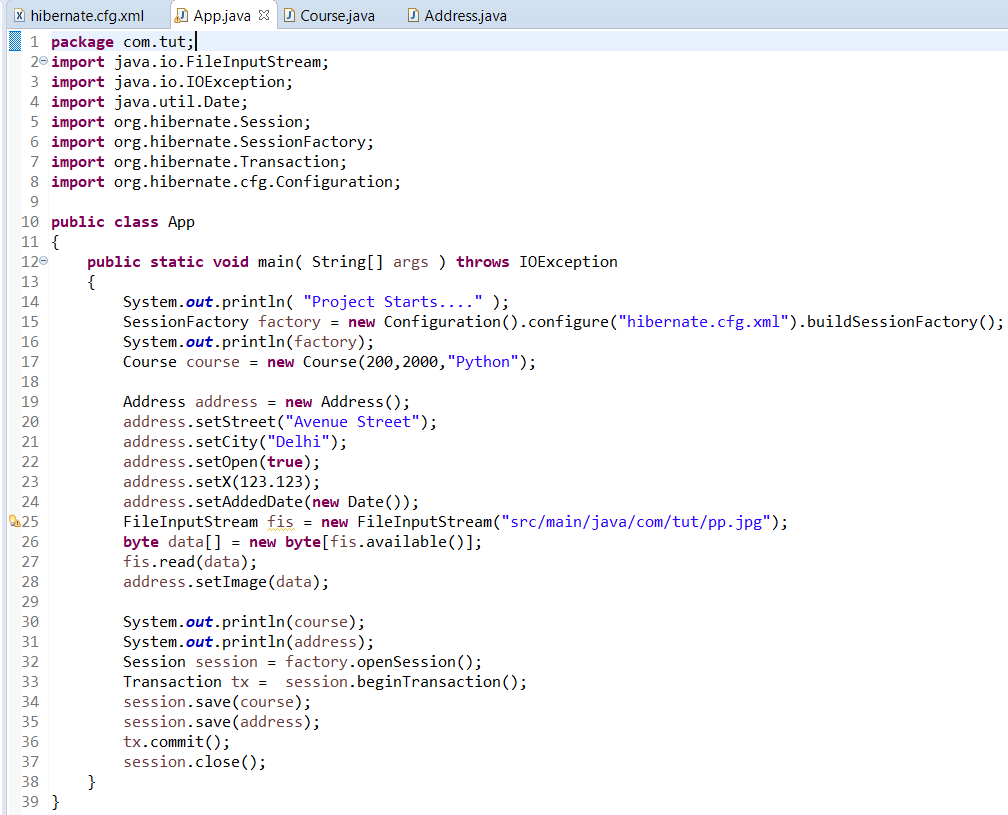


The dialect specifies the type of database used in hibernate

hbm2ddl.auto is a hibernate configuration property. It is used to validate and exports schema DDL to the database when the SessionFactory is created. Possible Values are “create”, “update”, “validate”, “create-drop”.

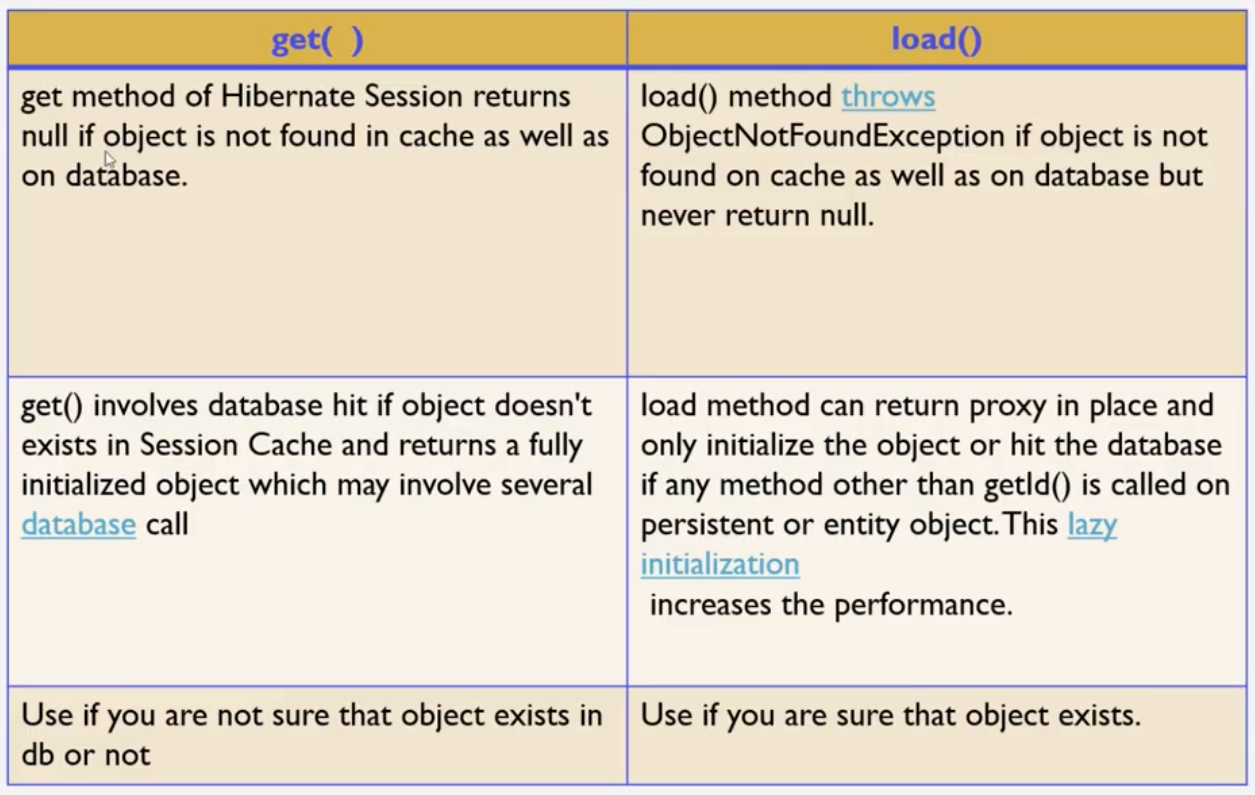


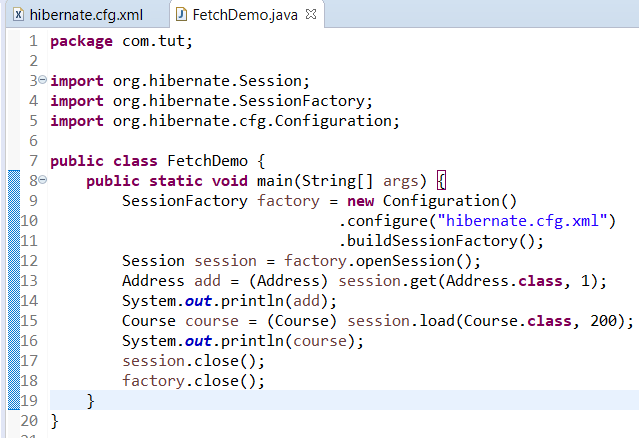
**Adding data including Image in database in Hibernate**



**FETCHING DATA using Hibernate.**

We can use any one of the get() method or load() method



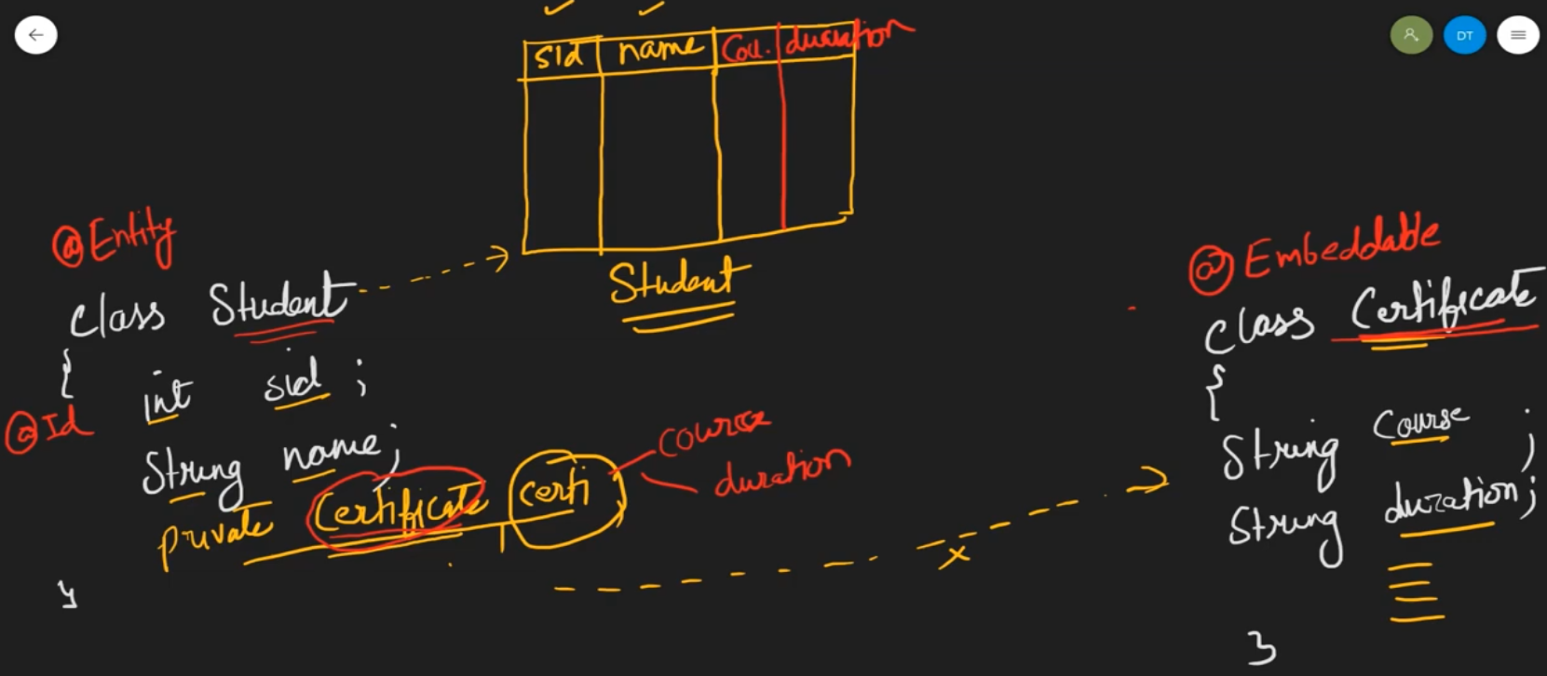


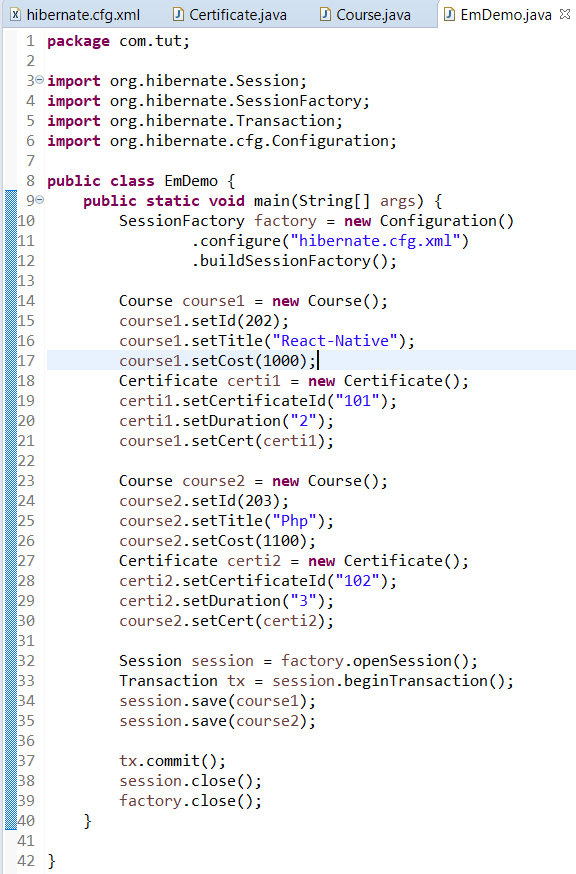
Hibernate.cfg.xml remains same in previous snapshot.

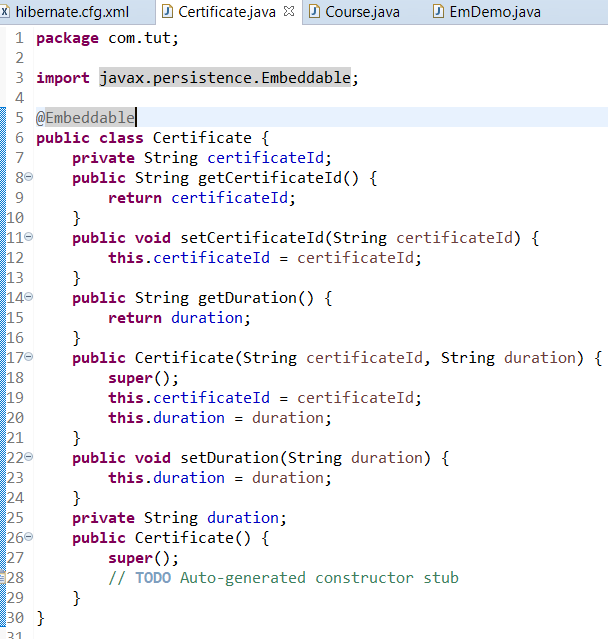
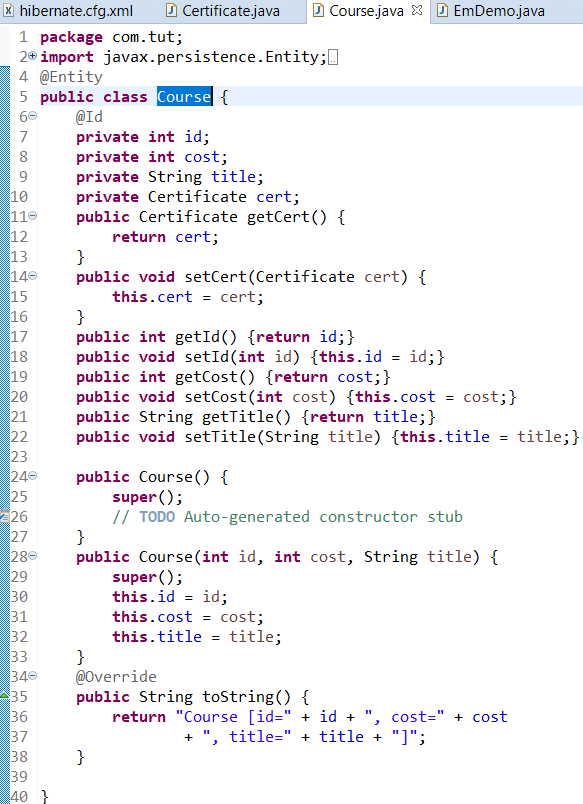
Example in video is different from Example actually implemented.

**@Embeddable in Hibernate**

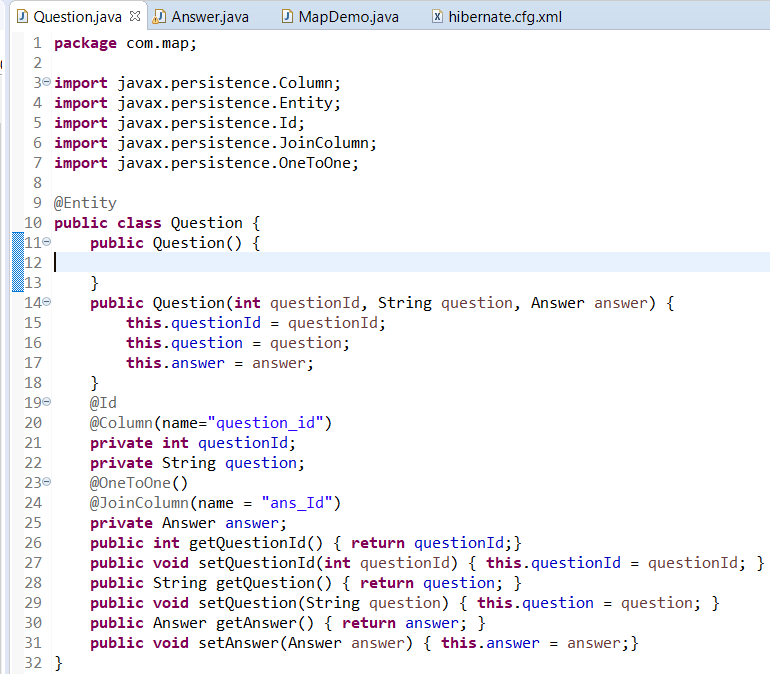
@Embeddable annotation tells Hibernate and any other JPA implementation that a class and its mapping annotations can be embedded into an entity. Example in video differ from example implemented.

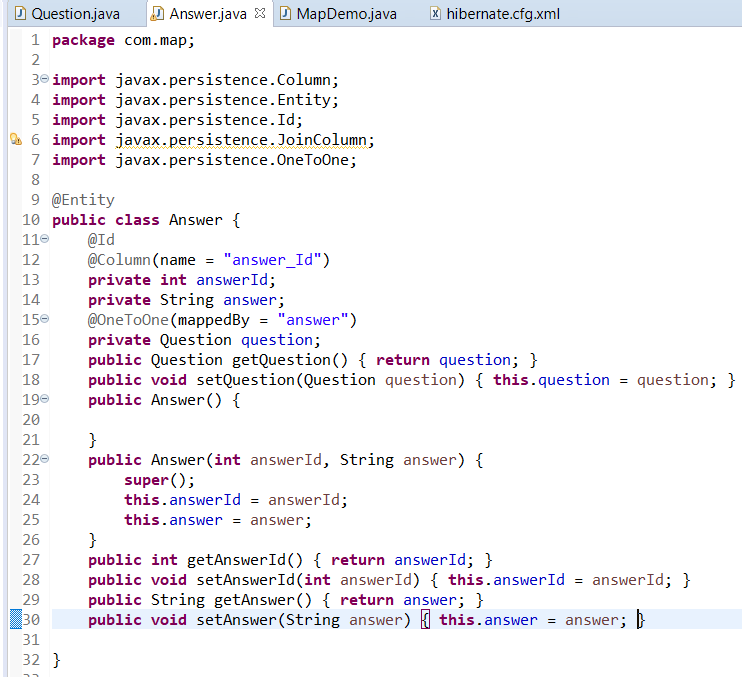


****

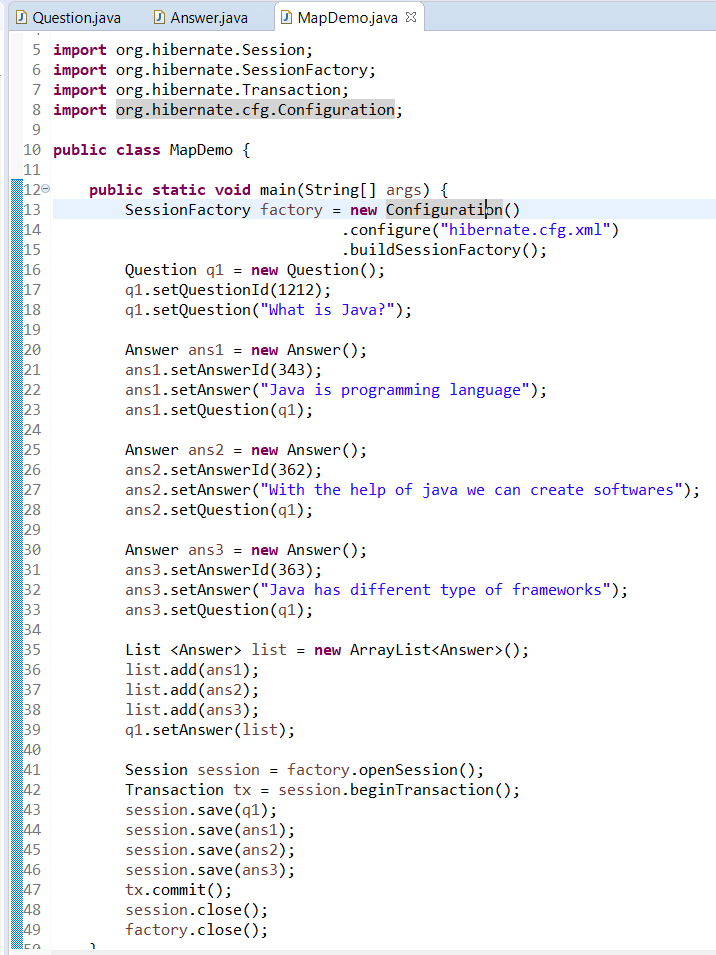
****

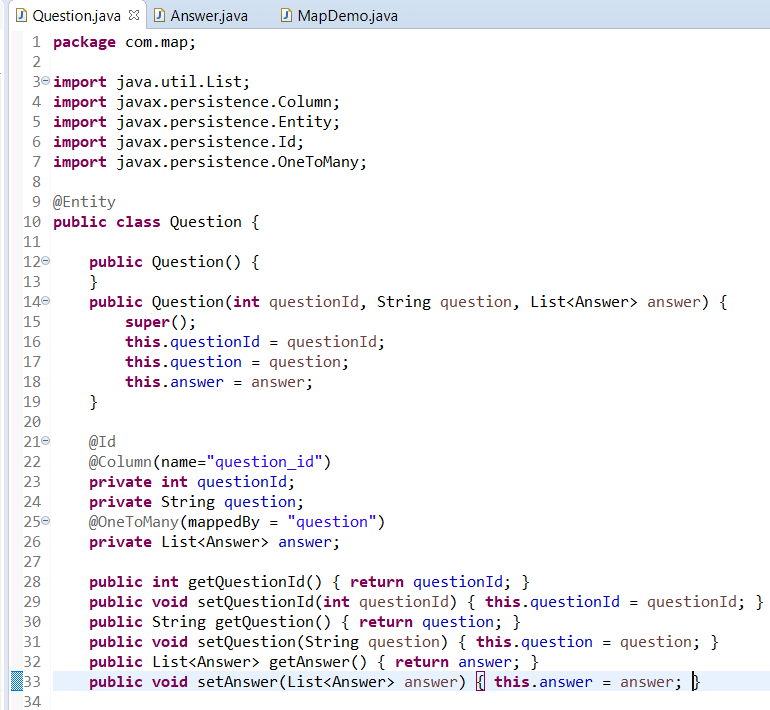
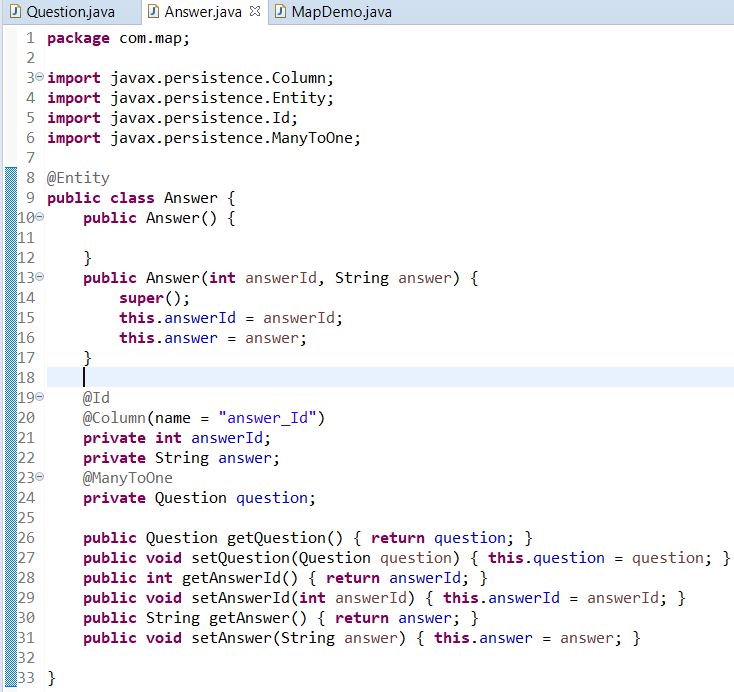
**OneToOne Mapping (uni-directional & bi-directional)**



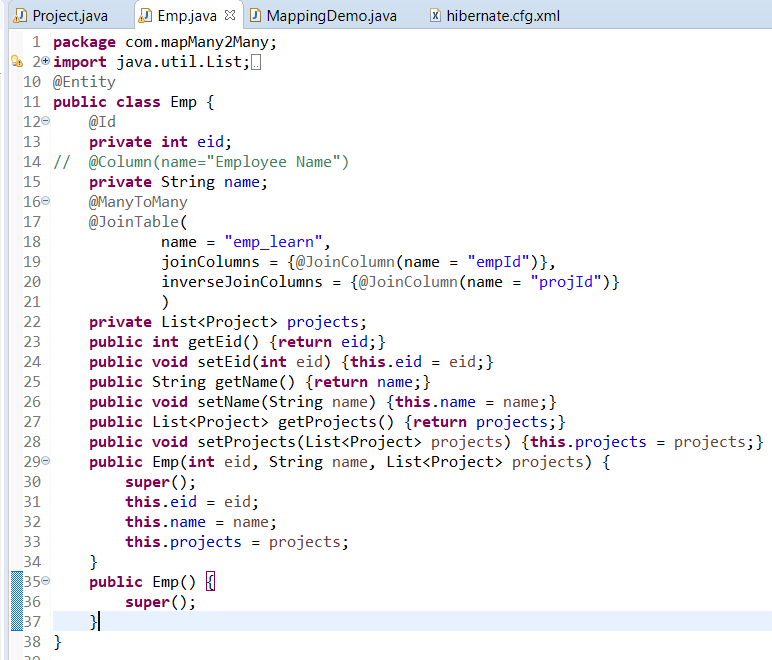
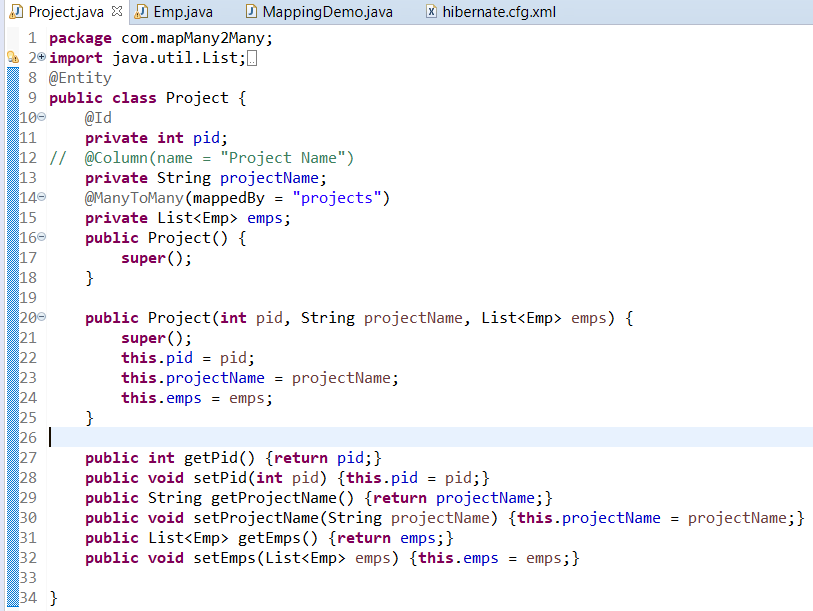


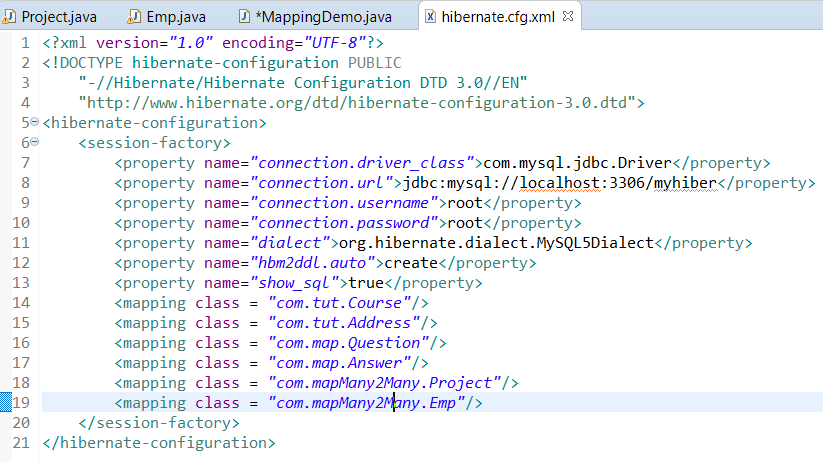
**@OneToMany Mapping**



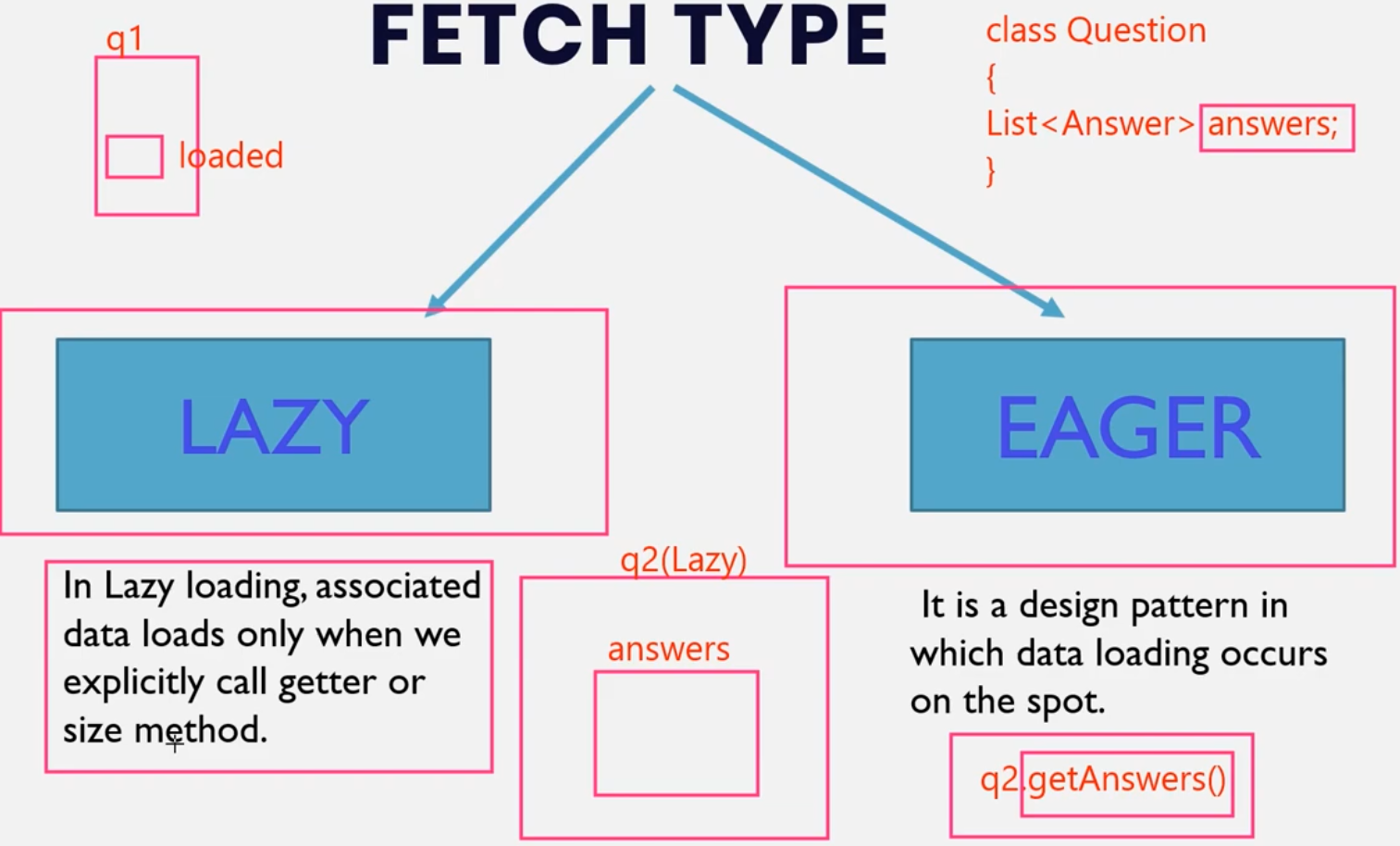


**@ManyToMany Mapping**





**Fetch Type In Hibernate**

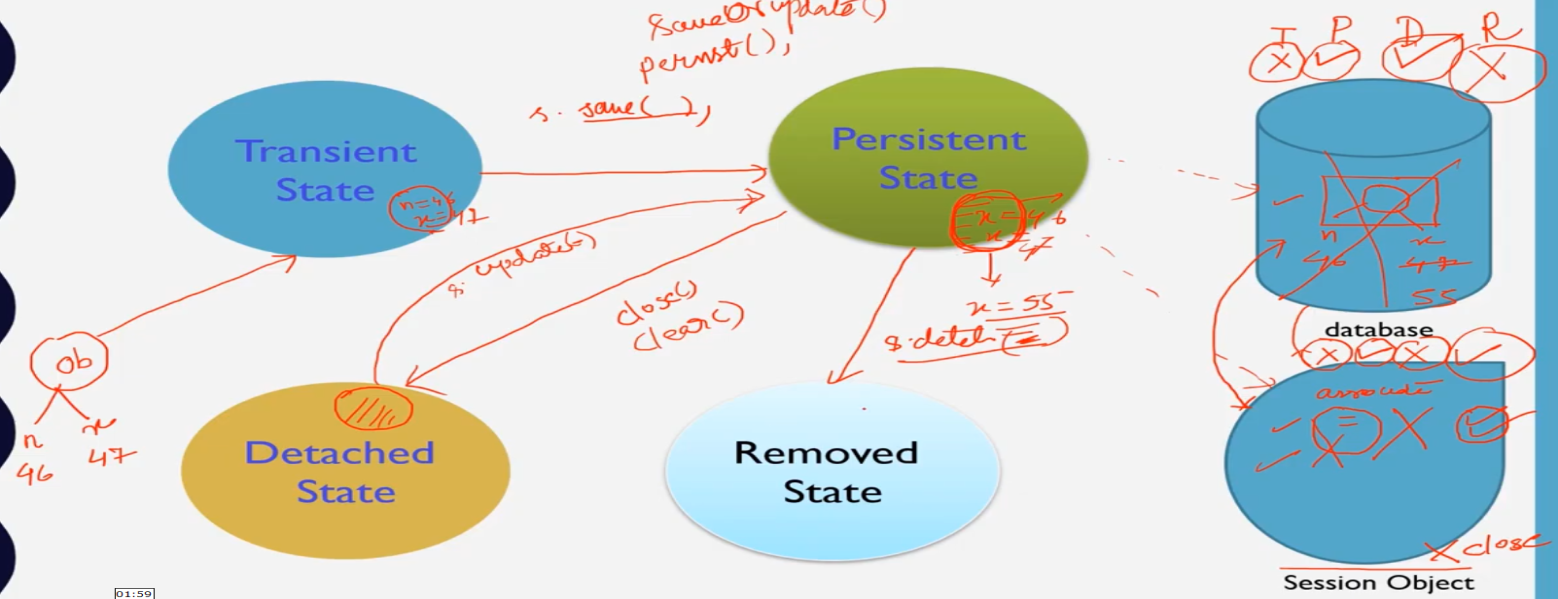
****

We will use the same example of ManyToOne Mapping

By default Lazy Loading is implemented. To use eager loading use

@OneToMany(mappedBy =”question”, fetch = FetchType.EAGER)

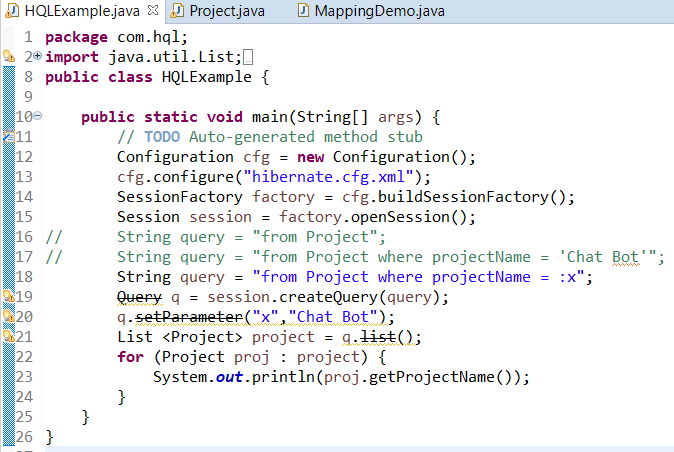
**Hibernate Object States (Persistence Life Cycle)**

****

Practical session not implemented

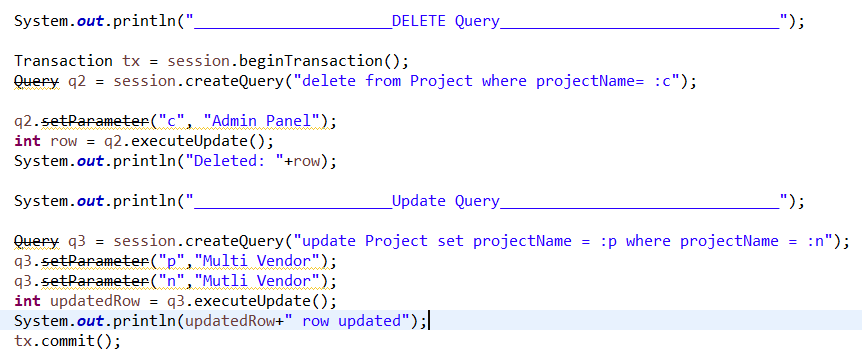
**HQL(Hibernate Query Database)**

Fetching data from database

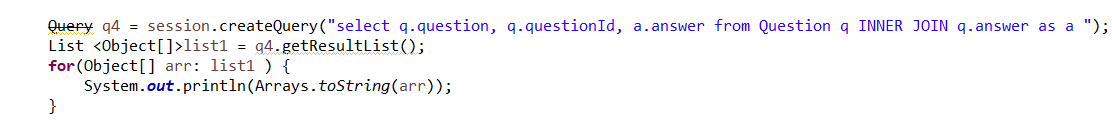


“Project” in line 16,17,18 is Name of Entity and “projectName” is the name of field in that entity.

**Update & Deleting data from database**

****

**Executing Join in Database**

****

In above snapshots we have used a “Query” which is depricated. To update it we just need to update the package fro which wuery is being imported.

Simply we need to change

**import org.hibernate.\*;** to

**import org.hibernate.query.\***;