**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. JPA determines what to do and how to do is determined by hibernate.

**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**.

**What is JPA?**

It is a programming interface that manages relational data in Java Application. Hibernate implements JPA.

NOTE : **When an object is "persisted," it means that an instance of a Java class (an entity) is saved or stored in a database.**

**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.  
Session Object is called as Persistence Object.

**What is SessionFactory**

SessionFactory provides an instance of Session.   
It 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.

*Configuration configuration = new Configuration();  
configuration.configure("hibernate.cfg.xml"); // Load Hibernate configuration from XML file  
SessionFactory sessionFactory = configuration.buildSessionFactory();  
Session session = sessionFactory.openSession();  
// Perform database operations using the session  
session.close(); // Close the session when done*

**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.
* @Access: used for defining the access type of either “FIELD” or “PROPERTY”. Default value is field.
* @Id - used to define primary key
* @Column - used to define column name.
* @EmbeddedId: indicates it is a composite primary key.

Generated Value, Transient,etc.

**How to create an immutable class in hibernate?**

Use @Immmutable annotation

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

a. It contains DB specific configurations, which is used to initilize sessionFactory.  
b. 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

If there are multiple databases, then seperate Session Factory has to be created for each database.

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) eg. Employee  
b. Create the mapping.  
c. Create the configuration file.  
d. Create class for creating and storing the persistent POJO.(The class annotated with @Entity)  
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.i.e Lazy Loading |
| 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 closes, 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.

**Mapping**

**One-to-One**

It is a situation in which an instance of an entity is associated with exactly one instance of another entity, and vice versa. It can be unidirectional & bidirectional.

In unidirectional, there is a one-way relationship from source entity to target another, and the target entity doesn't reference the source entity. In Bi-Directional vice versa also holds.

Example of Unidirectional one to one mapping:

@Entity  
@Table (name = "question")  
public class Question {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 @Column(name = "question\_id")  
 private Long questionId;

@Column(name = "question\_text")  
 private String questionText;

@OneToOne  
 @JoinColumn(name = "answer\_id", unique = true)  
 private Answer answer;  
// Constructors, getters, setters, and other properties  
}

@Entity  
@Table (name = "answer")  
public class Answer {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 @Column(name = "answer\_id")  
 private Long answerId;

@Column(name = "answer\_text")  
 private String answerText;  
// Constructors, getters, setters, and other properties

}

Example of Bi-Directional OnetoOne Mapping

@Entity  
@Table(name = "question")  
public class Question {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 @Column(name = "question\_id")  
 private Long questionId;

@Column(name = "question\_text")  
 private String questionText;

@OneToOne  
 @JoinColumn(name = "answer\_id", unique = true)  
 private Answer answer;

// Constructors, getters, setters, and other properties

}

@Entity  
@Table(name = "answer")  
public class Answer {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 @Column(name = "answer\_id")  
 private Long answerId;

@Column(name = "answer\_text")  
private String answerText;

@OneToOne(mappedBy = "answer")  
 private Question question;

// Constructors, getters, setters, and other properties

}

Note:

* Owning Side: It is the side which owns/controls foreign key column. It is present in the entity that defines the DB schema.
* Non Owning Side: It is the side which does not own/controls foreign key column.
* Mapped By will always be in non owning side. The value we specify for mapped by should be name of property in the owning side that establish the connection.

One-to-Many & Many-to-One: In this case an extra table may be created in case of bidirectional mapping. To avoid this we will use mappedBy

Example

@Entity  
@Table (name = "question")  
public class Question {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 @Column(name = "question\_id")  
 private Long questionId;

@Column(name = "question\_text")  
 private String questionText;

@OneToMany(mappedBy= “question”) 🡪 this prevent creating extra table “question\_answer”  
 private List<Answer> answers;  
// Constructors, getters, setters, and other properties  
}

@Entity  
@Table (name = "answer")  
public class Answer {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 @Column(name = "answer\_id")  
 private Long answerId;

@Column(name = "answer\_text")  
 private String answerText;

@ManyToOne  
@JoinColumn(“Question\_id”) 🡪 By default it is question\_question\_id. It creates FK.  
 private Question question;  
// Constructors, getters, setters, and other properties

}

Many-to-Many :

For this new table will be created always.

@Entity  
@Table (name = “Employee”)  
public class Emp{  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private String emp\_id;  
 private String emp\_name;  
 @ManyToMany  
 @JoinTable(name = “emp\_project\_table”,   
 joinColumns = {@JoinColumn(name = “Employee Id”)},   
 inverseJoinColumn = {@JoinColumn(name = “Project Id”)})  
 private List<Project> projectList;  
// constructors, getters and setters  
}

@Entity  
@Table (name = “Project”)  
public class Project{  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private String proj\_id;  
 private String proj\_name;  
 @ManyToMany  
 private List<Emp> empList;  
//constructors, getters and setters  
}

What does cascade attribute do?

Determines how saving, updating & delete operations should propagate from one entity to associated entities that are part of a relationship. There are multiple cascadeTypes(ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH).

What is Transaction.

Transaction is a logical unit of work that consists of one or more database operations (such as inserts, updates, or deletes) that are executed as a single, indivisible unit. Transactions are used to ensure the consistency, integrity, and reliability of the data in the database. The primary properties of a transaction are often referred to as the ACID properties.

* Atomicity: It means if any operation is performed on the data, either it should be executed completely or should not be executed at all. (All or Nothing)
* Consistency: Either the changed data passes all consistency check (foreign key or unique constraints) or all operations are rolled back and data is set back to the state we started.
* Isolation: It means if 2 transactions are happening concurrently, then these 2 transactions will not be visible to each other.
* Durability: It ensures that the data after the successful execution of the operation becomes permanent in the database.

What does @ Transactional do?

It is used to perform transaction management in spring boot. It can be applied at method level and class level. @Transactional annotation on the method in the same class takes precedence over @Transactional at class level.

* **Transaction Demarcation**: When a method with the @Transactional annotation is invoked, a new transaction is automatically started before the method's execution. This demarcates the beginning of a transaction scope.
* **Commit on Success**: If the annotated method completes successfully (i.e., no exceptions are thrown), the transaction is automatically committed, which means any changes made to the database within that transaction are saved.
* **Rollback on Exception**: If an exception is thrown within the annotated method, the transaction is typically rolled back automatically. This ensures that any changes made during the transaction are discarded, maintaining data consistency.
* **Nested Transactions**: Depending on the transaction management framework being used, @Transactional annotations can support nested transactions. Nested transactions allow you to create a hierarchy of transactions, where inner transactions can be rolled back independently of outer transactions.
* **Customization**: The @Transactional annotation often supports various attributes that allow you to customize transaction behavior. You can specify attributes like isolation levels, propagation behaviors, timeout values, and more depending on the framework being used.
* **Declarative Approach**: @Transactional enables a declarative approach to transaction management. You don't need to write explicit transaction handling code; instead, you annotate methods or classes to indicate the transactional behavior you desire.

Here's a simple example of using @Transactional in Spring:

@Service  
@Transactional  
public class MyService {  
 @Autowired  
 private MyRepository myRepository;  
 public void performDatabaseOperation() {  
 // This method runs in a transaction  
 myRepository.saveEntity(entity);  
 }  
}

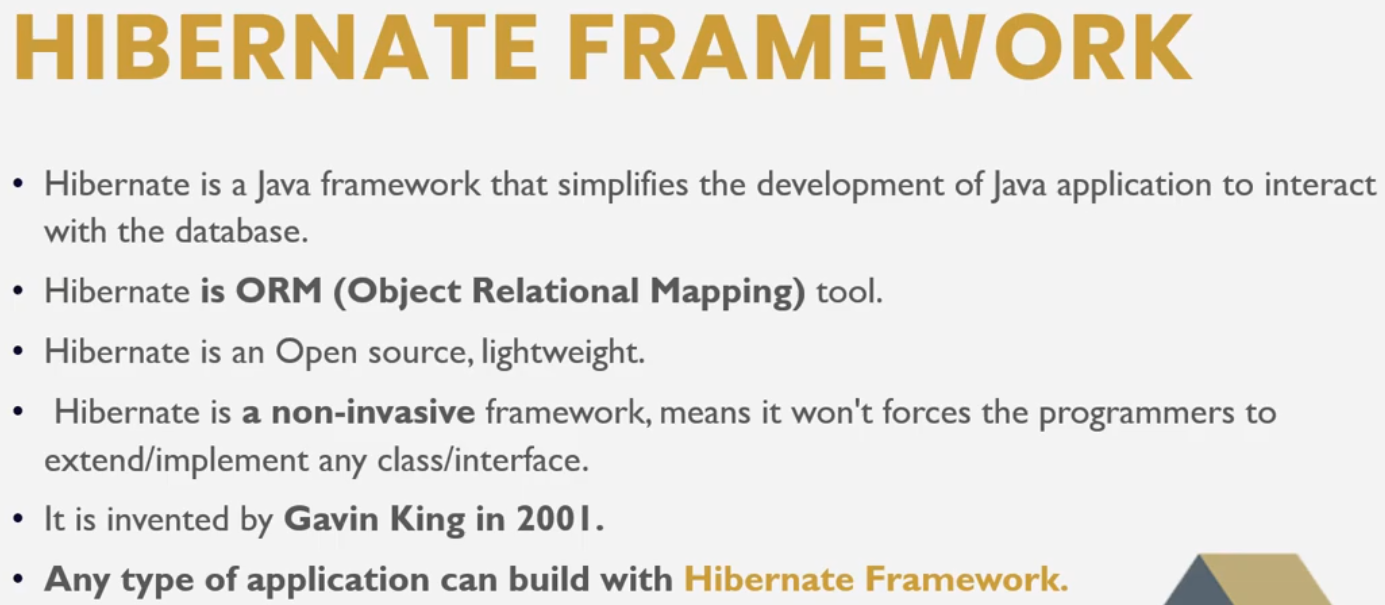
In this example, the @Transactional annotation on the MyService class ensures that the performDatabaseOperation method runs within a database transaction. If the method completes successfully, the transaction is committed, and any changes made to the database are saved. If an exception occurs, the transaction is rolled back to maintain data consistency.

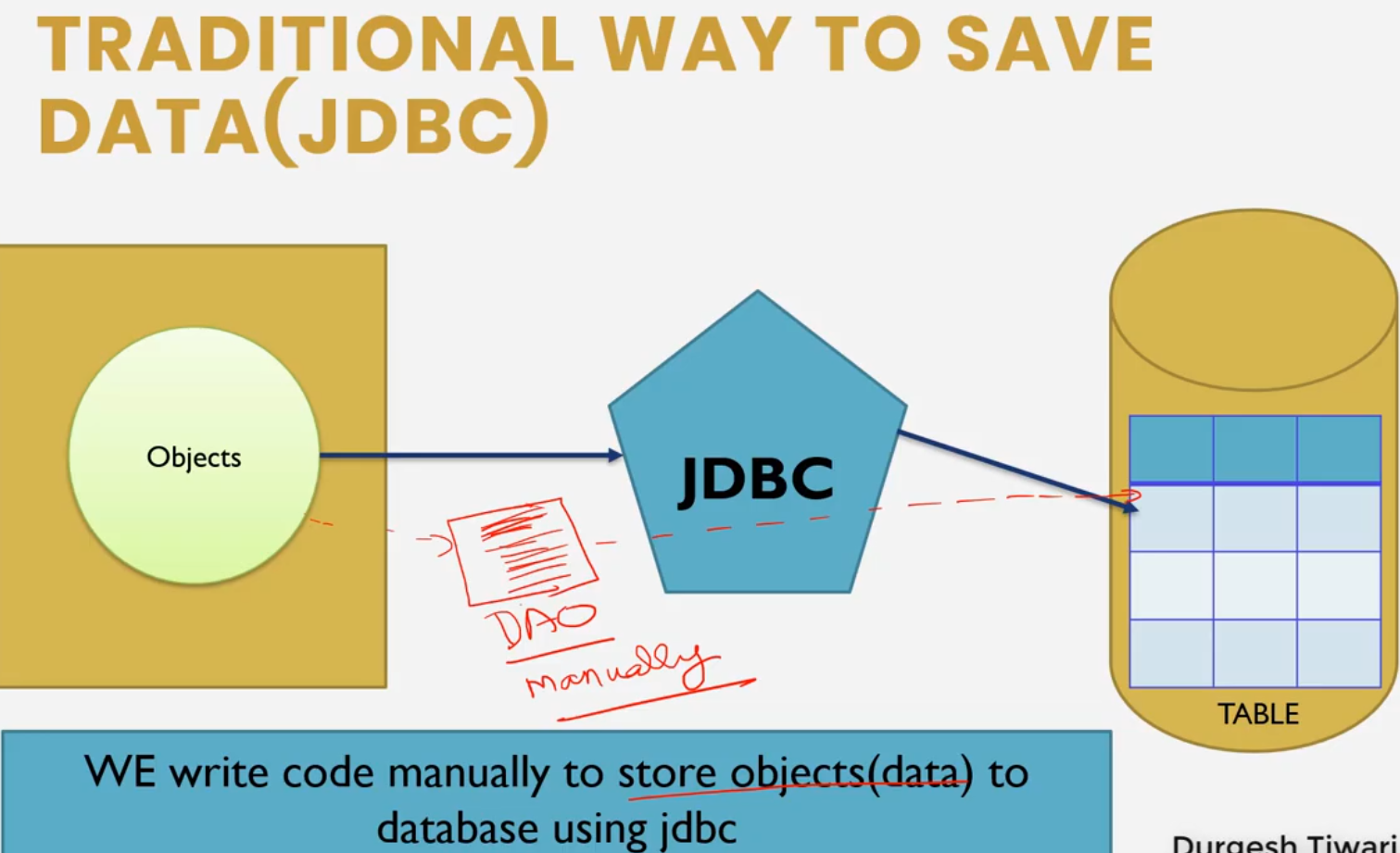
**Transaction Propagation**

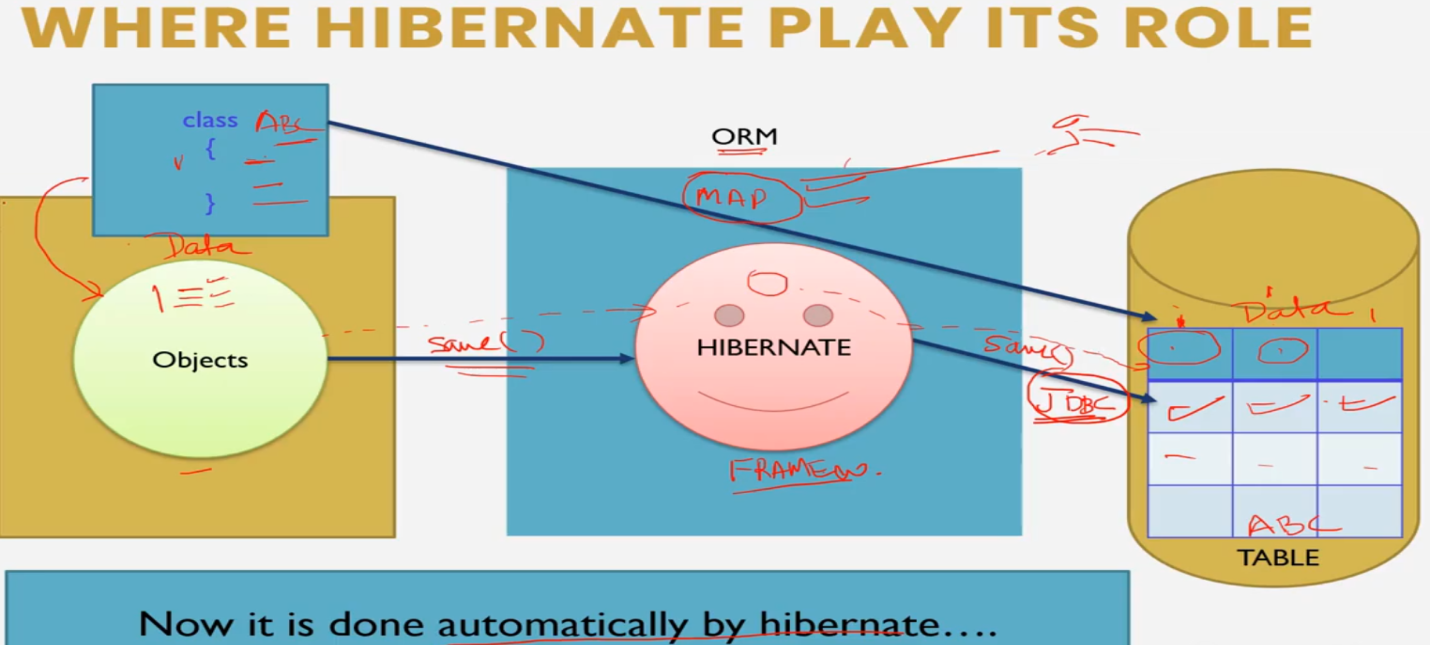
In the context of the @Transactional annotation in Java frameworks like Spring, "transaction propagation" refers to the behavior that dictates how an annotated method should interact with an existing transaction or start a new one if none exists. Transaction propagation defines the rules for how transactions should be handled when a method annotated with @Transactional is called, especially in scenarios involving nested method calls.

There are several transaction propagation behaviors available in the @Transactional annotation, each specifying how a method should participate in the current transaction context. The most commonly used propagation behaviors include:

* **REQUIRED**: This is the default propagation behavior. If a transaction context already exists when the method is called, the method will join that transaction. If no transaction exists, a new one will be created.
* **REQUIRES\_NEW**: This behavior always starts a new transaction, suspending the current one if it exists. It ensures that the method's work is isolated within its own transaction.
* **SUPPORTS**: This behavior doesn't start a new transaction but participates in an existing one if available. If no transaction exists, the method executes without a transaction.
* **MANDATORY**: This behavior requires that an existing transaction is available; otherwise, it throws an exception. It enforces that the method must be executed within an existing transaction context.
* **NOT\_SUPPORTED**: This behavior suspends the current transaction if one exists and executes the method without a transaction context. It ensures that the method is executed independently of any ongoing transactions.
* **NEVER**: This behavior ensures that no transaction context is available when the method is executed. If a transaction exists, it throws an exception.
* **NESTED**: This behavior allows for nested transactions within the scope of the current transaction. If a transaction exists when the method is called, it starts a new savepoint within that transaction. If no transaction exists, it behaves like REQUIRED.





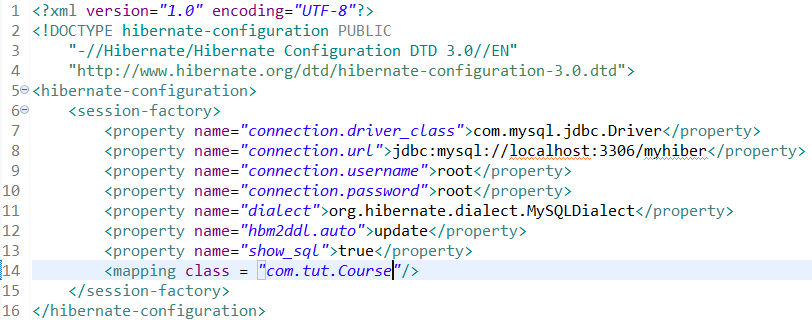


* Add Hibernate Core and Mysql Connector dependency in POM.xml

**Adding hibernate Configuration**

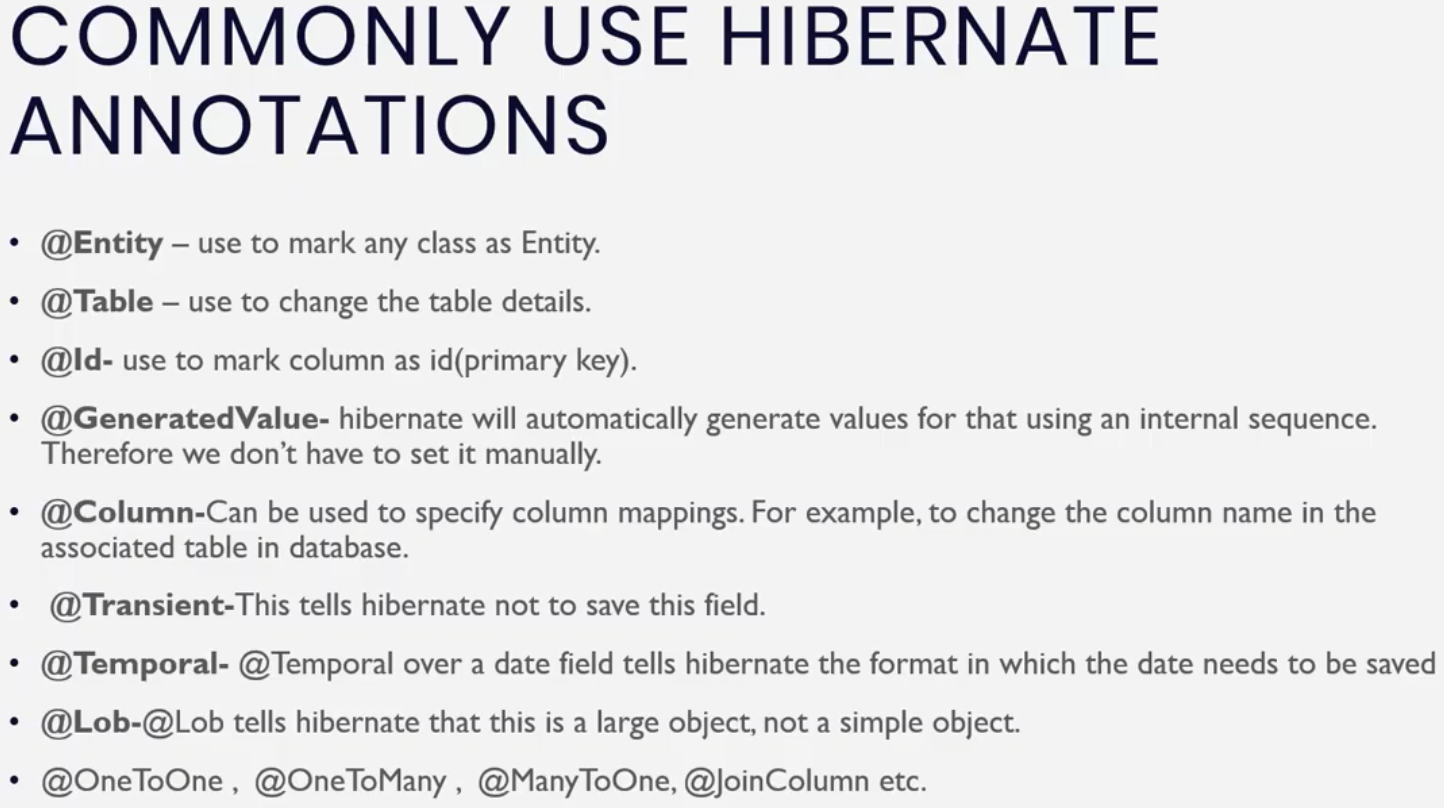
Right click on src/main/java add a xml fle hibernate.cfg.xml

Add the following dtd configuration file

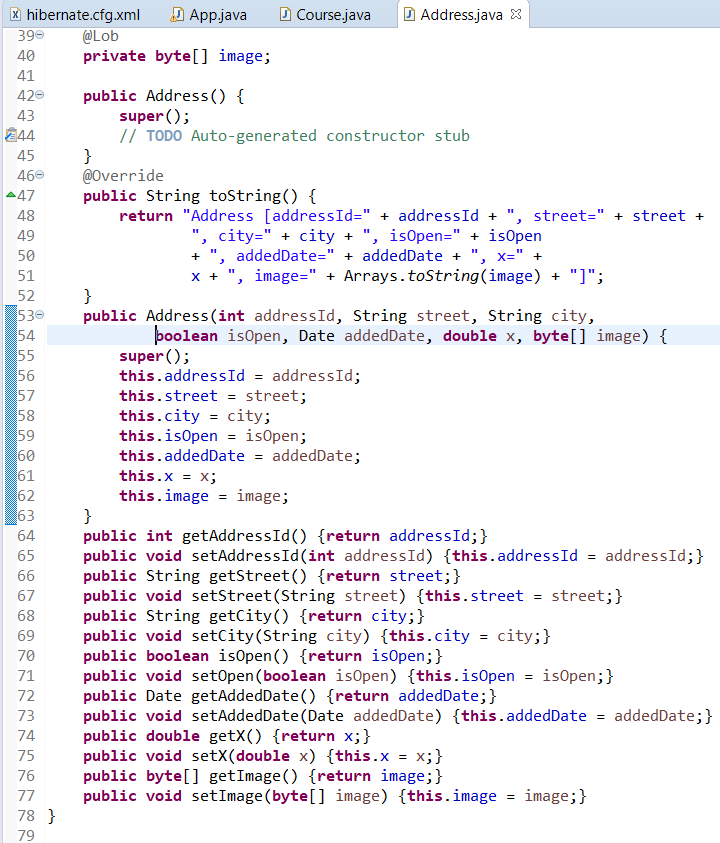
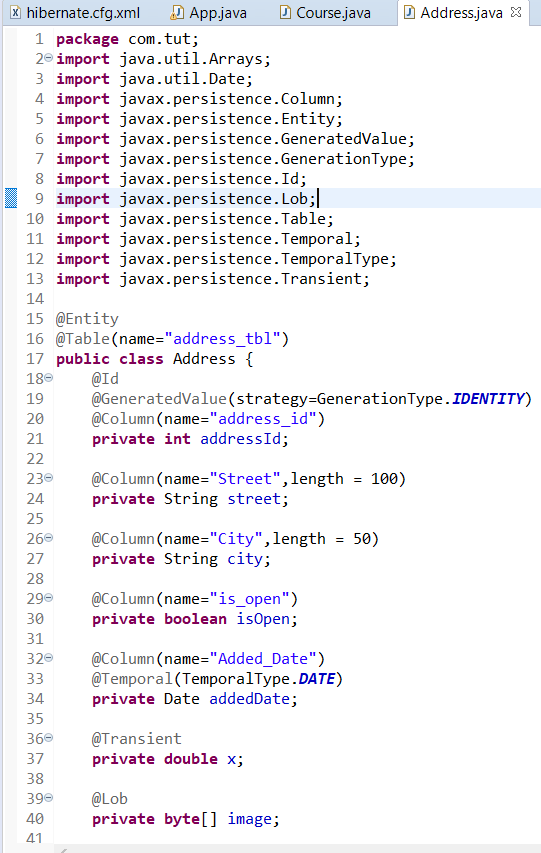
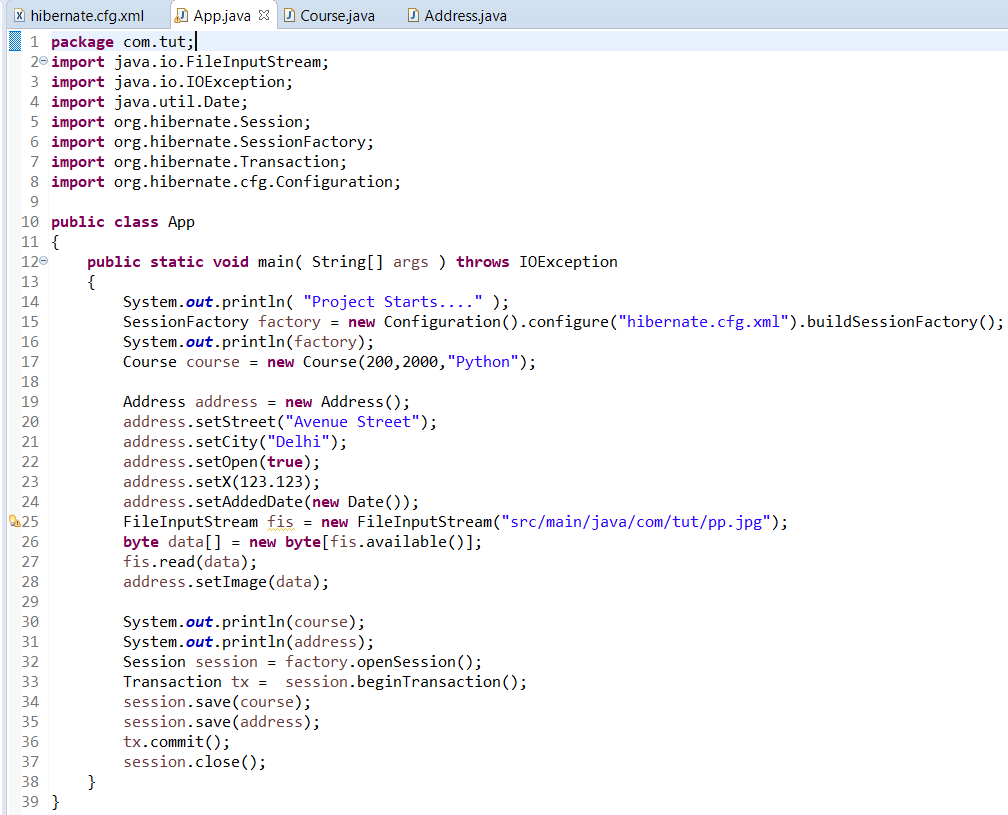


The dialect specifies the type of database used in hibernate

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”.

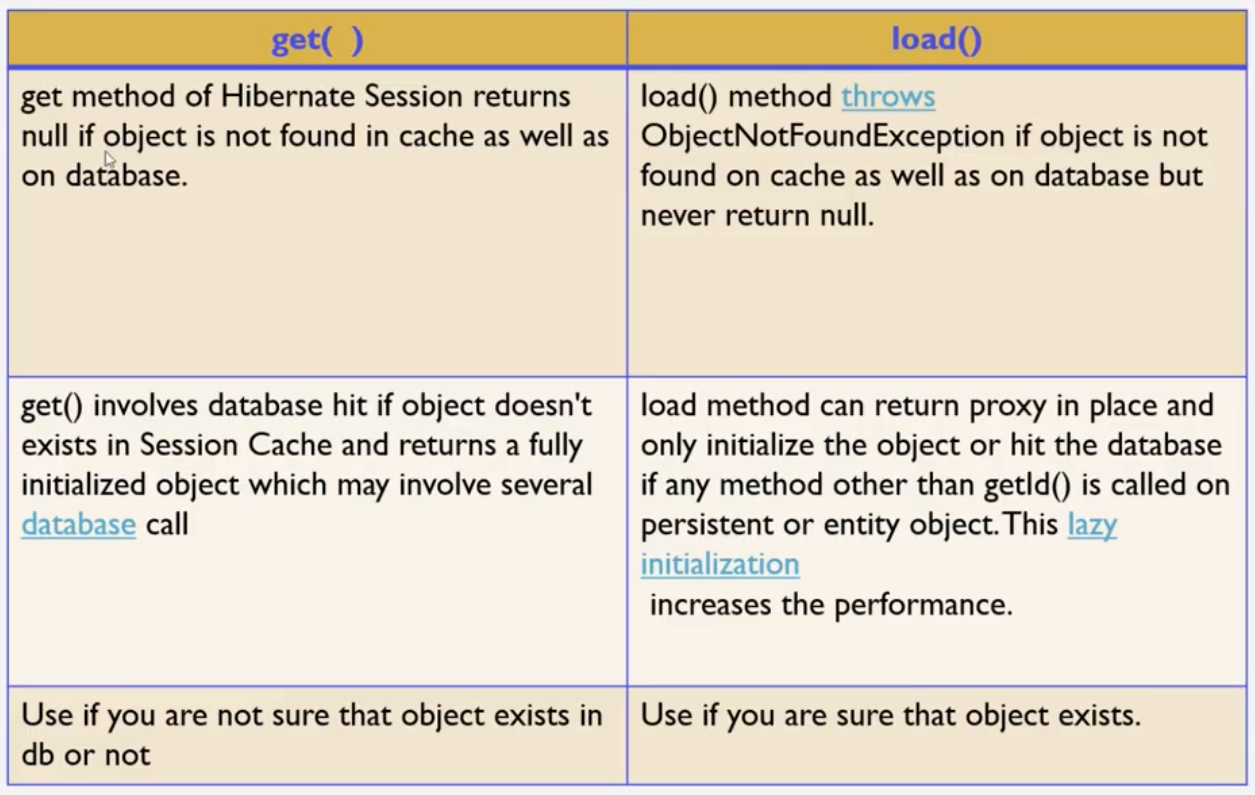


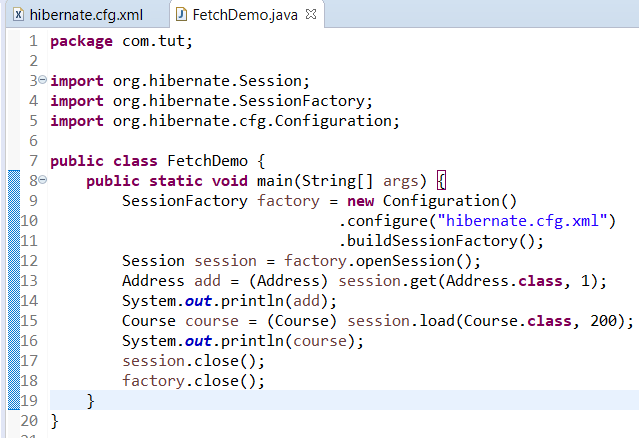
**Adding data including Image in database in Hibernate**



**FETCHING DATA using Hibernate.**

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



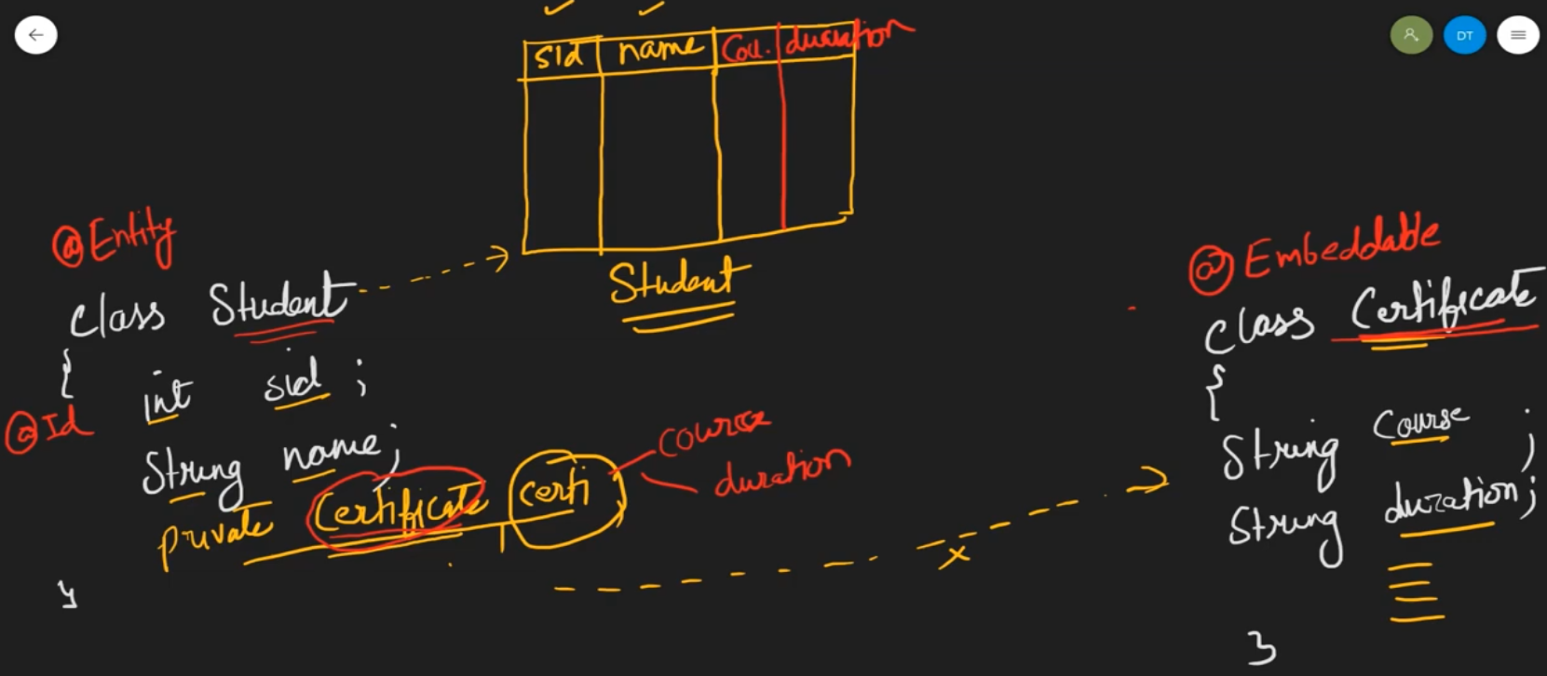


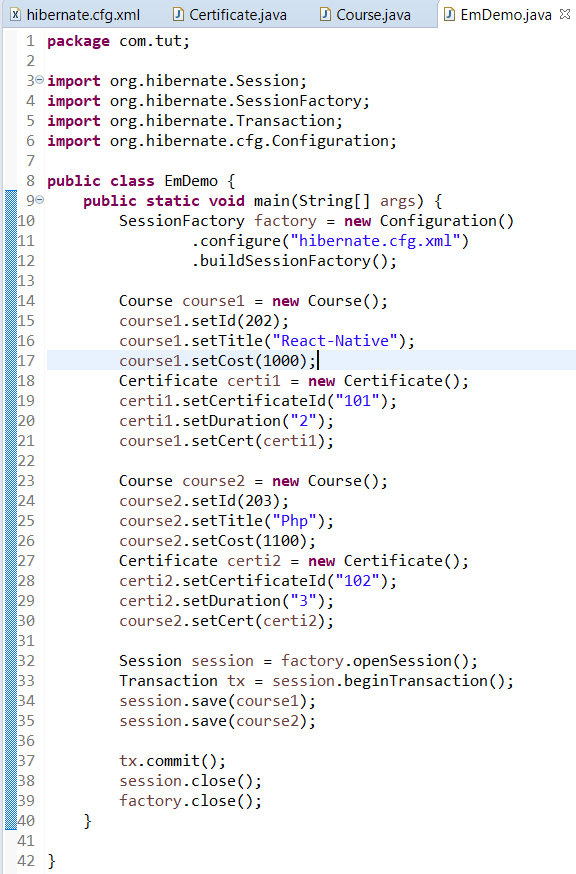
Hibernate.cfg.xml remains same in previous snapshot.

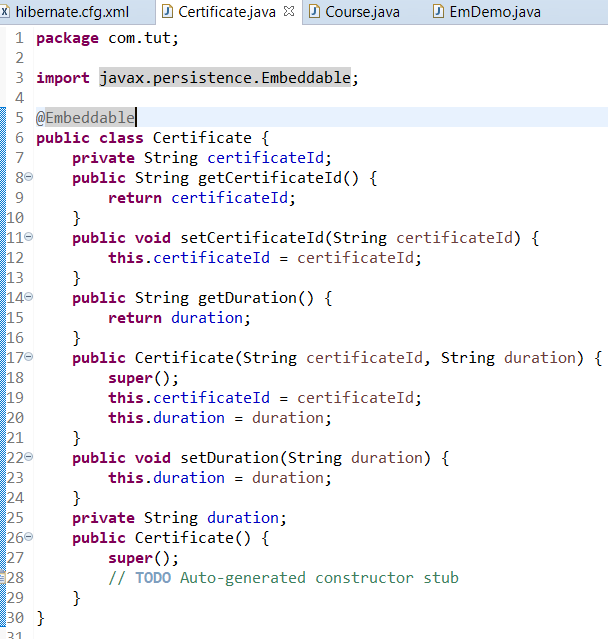
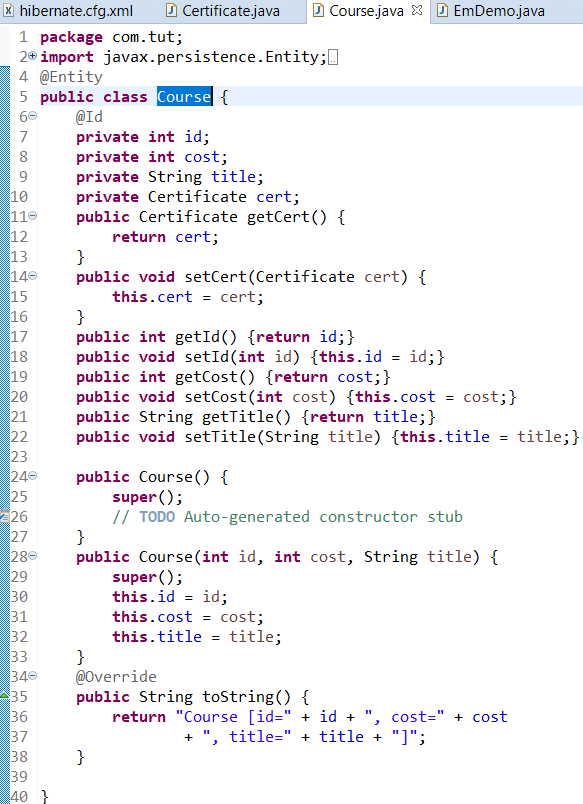
Example in video is different from Example actually implemented.

**@Embeddable in Hibernate**

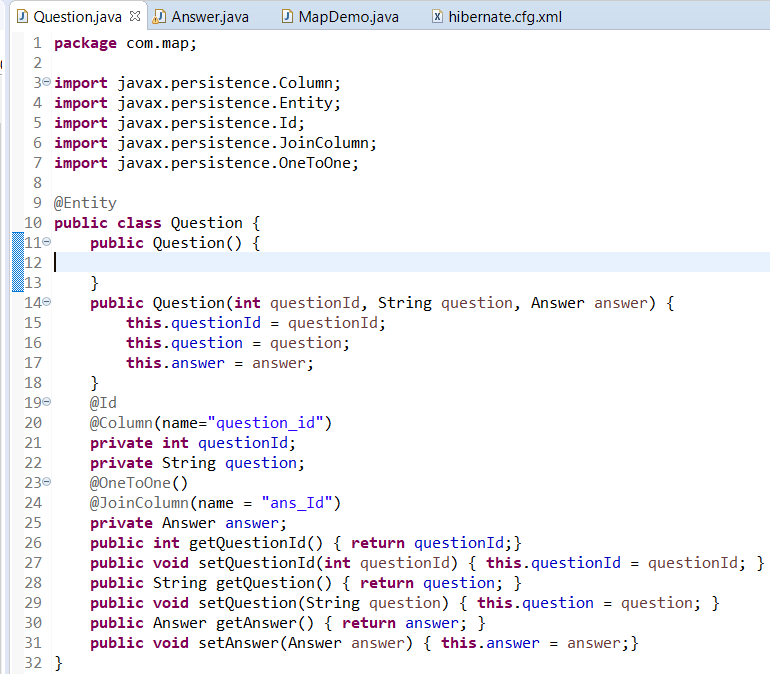
@Embeddable annotation tells Hibernate and any other JPA implementation that a class and its mapping annotations can be embedded into an entity. Example in video differ from example implemented.

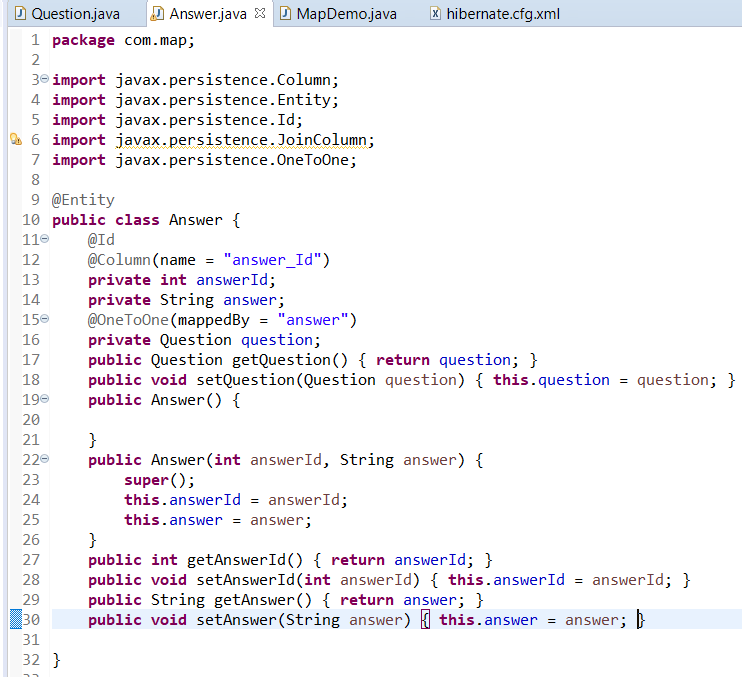


****

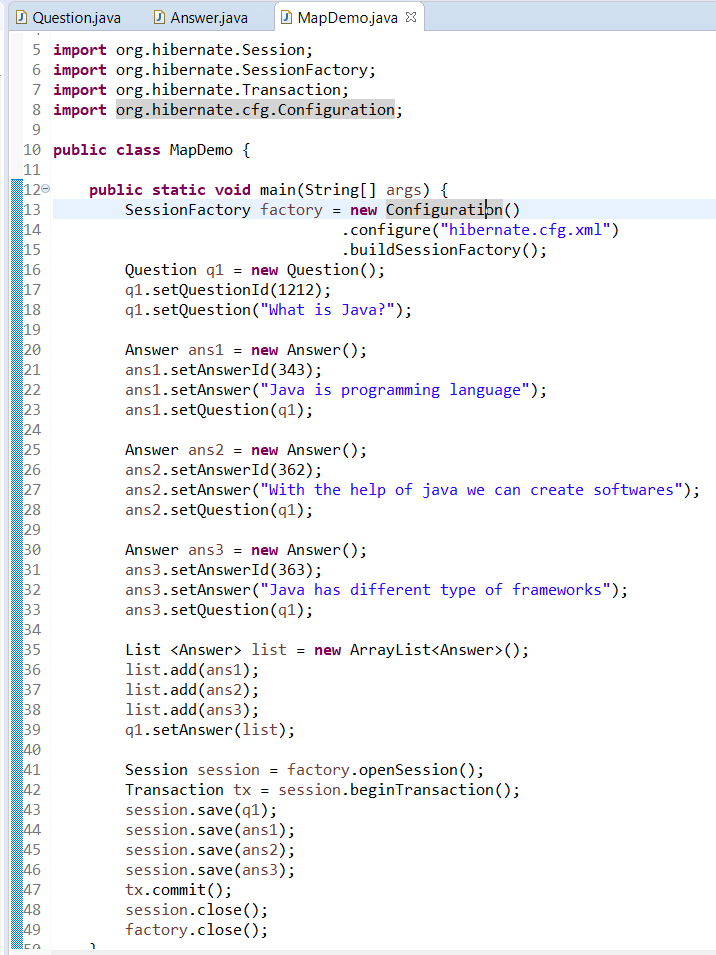
****

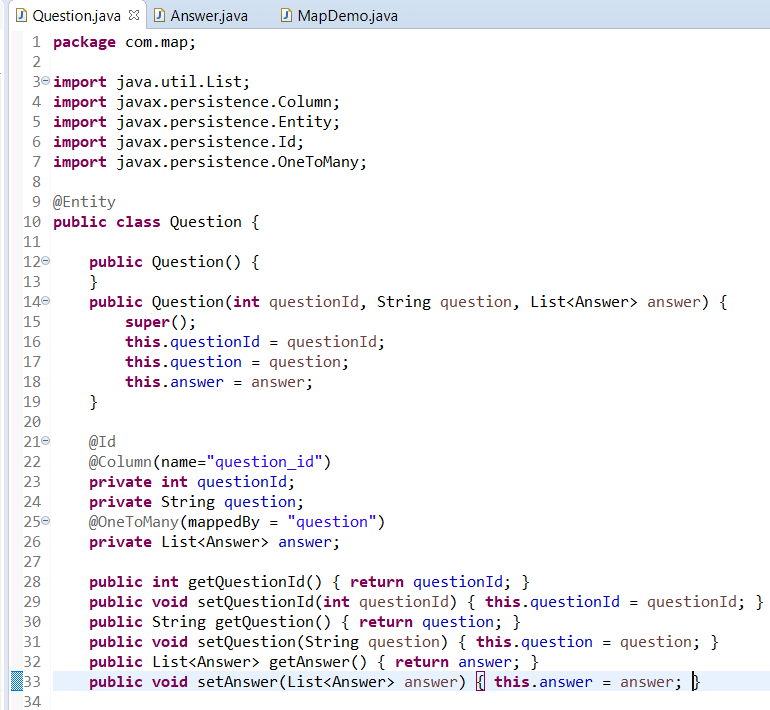
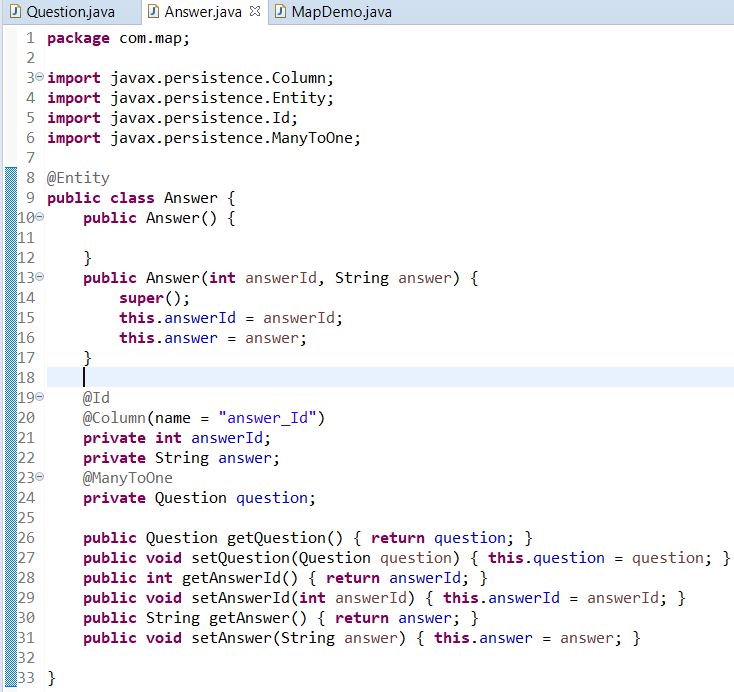
**OneToOne Mapping (uni-directional & bi-directional)**



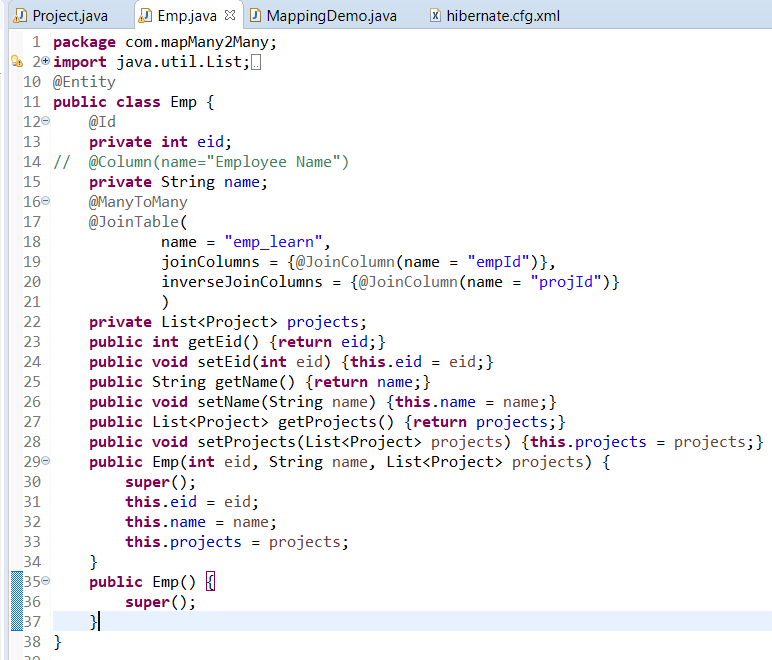
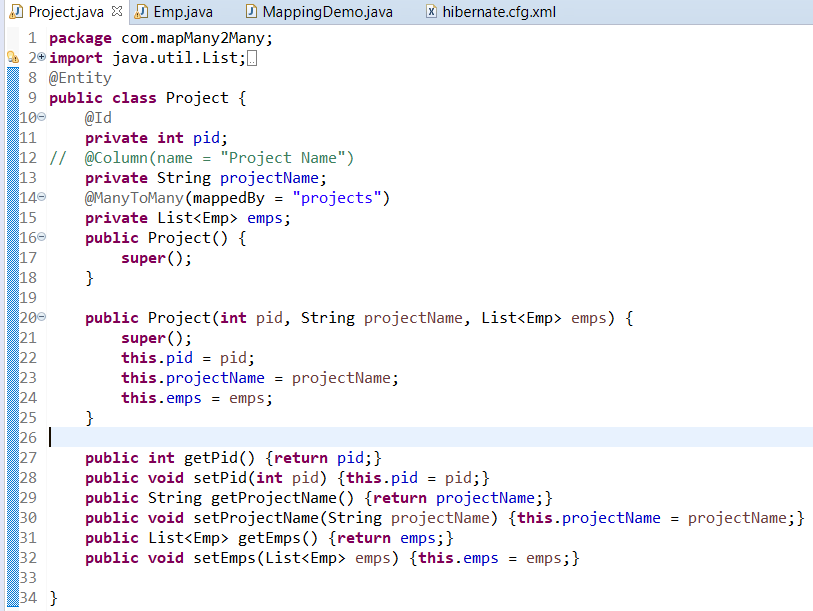


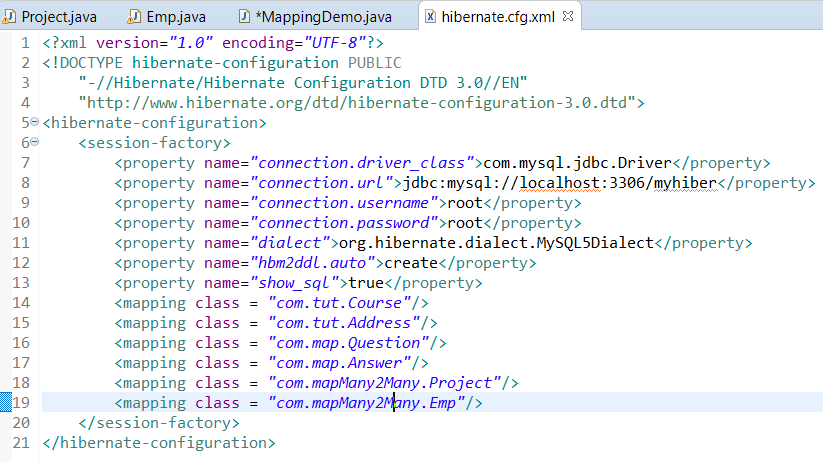
**@OneToMany Mapping**



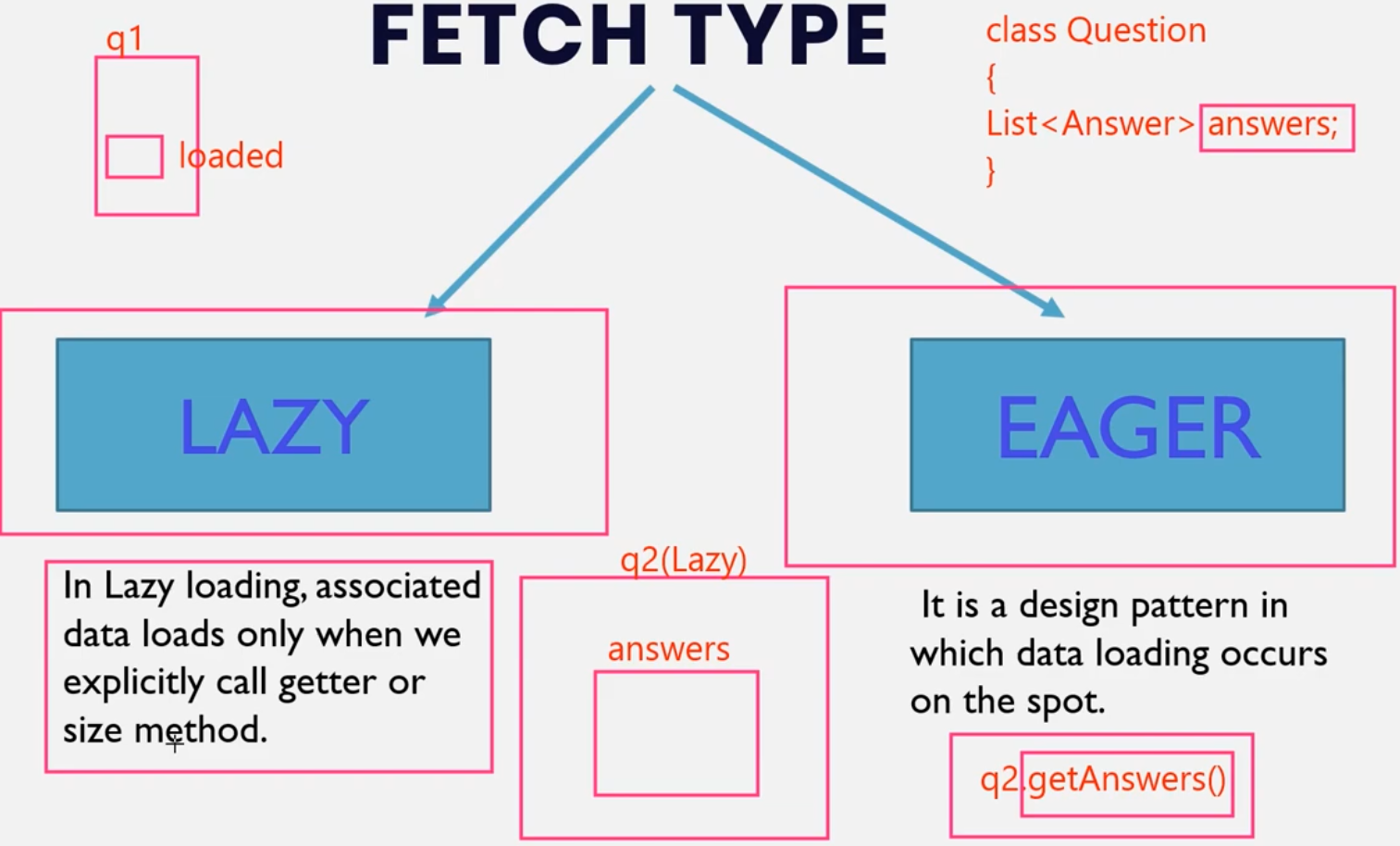


**@ManyToMany Mapping**





**Fetch Type In Hibernate**

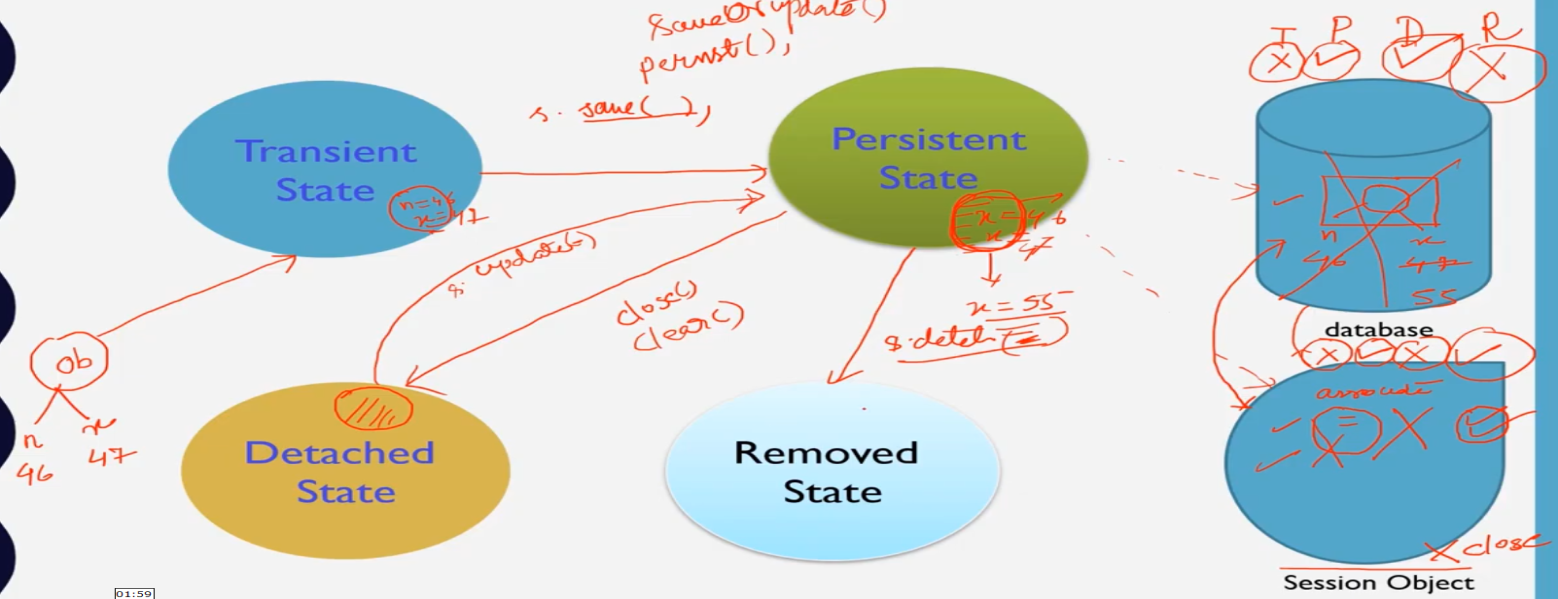
****

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)

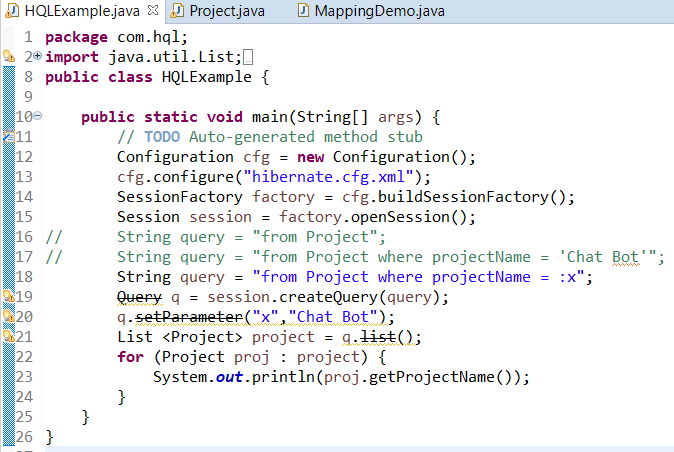
**Hibernate Object States (Persistence Life Cycle)**

****

Practical session not implemented

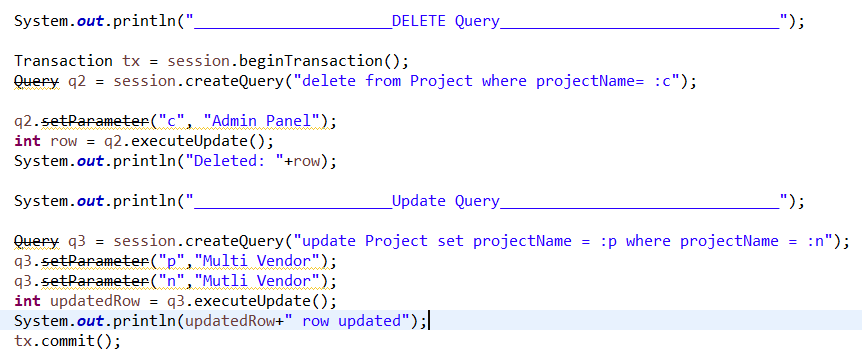
**HQL(Hibernate Query Database)**

Fetching data from database

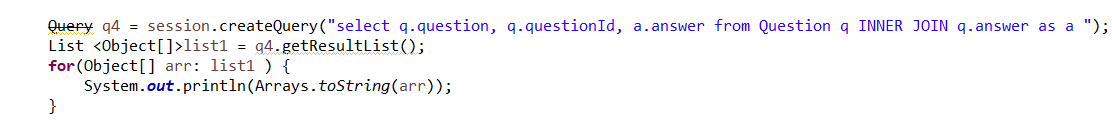


“Project” in line 16,17,18 is Name of Entity and “projectName” is the name of field in that entity.

**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.\***;