Frameworks

Date : 09/04/24

* Drawbacks of JDBC API :-

- While writing a JDBC program, the programmer is forced to take care of SQL related properties and credentials.

- SQL queries need to be written explicitly by the programmer.

- In JDBC, the conversion from JAVA class to the database table is not possible. There is no relationship between the JAVA objects and data in the database.

- To perform JDBC operation, we need to write lengthy code.

- We need to follow 5 steps everytime to perform CRUD operation.

Due to the above drawbacks, JDBC is a depricated technology.

ORM

* ORM

- ORM stands for object Relational Mapping.

- It is a technology for converting JAVA objects to the relational databases.

JDBC - 5 steps, db dependent, sql queries, No relation b/w java object, db & tables explicitly, complex multiple queries

Class → table
Variable → columns
object → rows

Date :

- ORM internally impliments JDBC. It was developed to overcome the drawbacks of JDBC. With the help of ORM, the programmer doesn't need to worry about database related properties and no need to write SQL queries on our own.

- With the help of ORM, the classes in JAVA will be converted into tables in the database, the properties of the clases are converted to columns of the table and the objects created for the clases are converted into rows in the table.

- **Advantages of ORM :-**

- It saves time and effort.
- It makes development more object-oriented.
- It reduces the development cost.
- No need to implement database manually.

- **ORM Tools :-**

i) Hibernate
ii) OpenJPA
iii) Eclipselink
iv) Toplink
v) MyBatis (iBatis)

Hibernate

Date :

* Hibernate :- (version used :- 5.0)

- Hibernate is a ORM tool and JAVA persistence framework that simplifies the development of JAVA application to interact with the database.

- It is an open source, widely used, light-weight ORM tool.

- The performance of Hibernate is faster because cache is internally used.

- Hibernate implements specification of JPA (Java Persistence API) for data persistence (CRUD opn).

• JPA :-

- Java Persistence API is a specification of JAVA used to persist data between Java object and relational database.

- It acts as a bridge between object-oriented domain models and relational database system.

• Diff^n b/w Hibernate and JPA

→ S.L.E (Self learning Exercise)

Specification (abstract)	Implementation
Java Persistance Query language	Hibernate Query Language HQL
bridge b/w java & JDBC	org.hibernate
javax.persistant pack	

Date : 10/04/24

- To connect JAVA appⁿ with database, JPA provides 3 interfaces :-
  i) EntityManagerFactory
  ii) EntityManager
  iii) EntityTransaction

i) EntityManagerFactory :-

- It is a interface present in javax.persistant package. It is used to provide EntityManager.
- It provides an efficient way to construct multiple EntityManager instances for the database

efficient way to obtain multiple instances of

ii) EntityManger :-

- It is an interface present in javax.persistant package which manages the lifecycle of an entity instance.
- It also provide methods to do CRUD operation with the database

iii) EntityTransaction :-

- It is an interface present in javax.persistant package which provides methods to handle transaction in JPA application
- Transaction is set of operation that either fail or succeed.
- A database Transaction consist of set of SQL operations that are Commited or Rolled back

POJO → Plain Old Java Object
↳ only getter/setter

Persistant unit name - PUN

Bean →

Date :

* Methods of Entity Manager

i) persist(Object entity):-

- It is used to insert the data into the database

ii) merge(object entity):-

- It is used to update the data in the database based on primary key.

iii) remove(object entity):-

- It is used to delete a particular row based on primary key.

iv) find(class<T> entityclass, object primary key):-

- It is used to fetch the data from the database using primary key.
- It gives single row as output.

- persistance.xml

- It is a standard configuration file in JPA.
- It has to be included META-INF directory.

src/main/resources
↳ META-INF (folder)
↳ persistance.xml (file)

project management tool - maven project

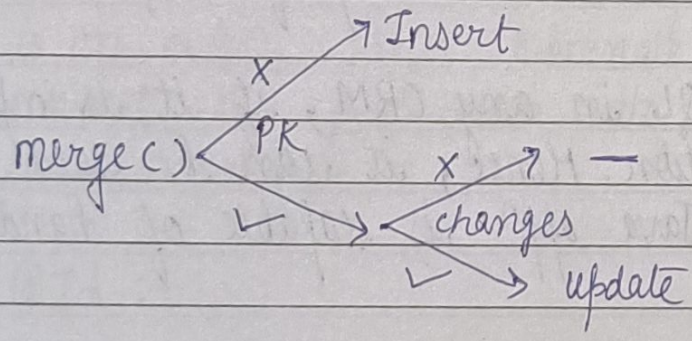
Date :

- The persistence.xml file must be define a persistence-unit with a unique name.
- The persistence.xml file is used to configure entity clases, provide database driver, database connection information and other relational mapping details.

- pom.xml :-

- It is an xml file that contains information about the project and configuration details used by maven to build the project

- polymorphism Ex →



polymorphic nature of merge method

- Note :-

merge() is polymorphic in nature. If the primary key is present, it acts like update query. If the primary key is not present in the table, data will be inserted ie. it acts like insert query.

Date : 11/04/24

* JPQL

- JPQL stands for Java Persistence Query Language.
- Hibernate writes its own queries in HQL (Hibernate Query Language). If the app" demands change in the ORM tool, the other ORM tool will not be working on HQL. Hence, it would make it difficult to migrate from one ORM tool to another.
- Example :- The MyBatis ORM works on DQL (Dynamic Query Language).
- In Order to make it easy to migrate from one ORM to another, we will take help of JPQL.
- We can use JPQL in any ORM, if it is implemented in Java application. Hence, it can be said any ORM used in Java app" is capable of handling JPQL.
- For implementing JPQL, in JPA application, we need to take help of an interface called "Query".

• Note :-

- In JPQL, we do not mention the table name & column name. Instead we use class name and property name respectively.

Diff^n b/w JPQL & SQL

Date :

SQL	JPQL
- SELECT * FROM EMPLOYEE	SELECT e FROM Employee e
- SELECT * FROM EMPLOYEE WHERE PHONE = ?	SELECT e FROM Employee e WHERE e.phone = ?1

* Annotations of JPA (`javax.persistence`)

i) @Entity :-

- This annotation is used to specify that the class is an entity. When hibernate (ORM tool) looks at the annotation it will convert the java class into a table in the specified database.

ii) @Id :-

- This annotation is used to specify the primary key of the entity.
When hibernate (any ORM tool) looks at this annotation it will convert the column into the primary key.

iii) @Table :-

- This annotation is used to control and modify the properties of the table.

Ex :- To change the name of table :- @Table (name="value")

Date :

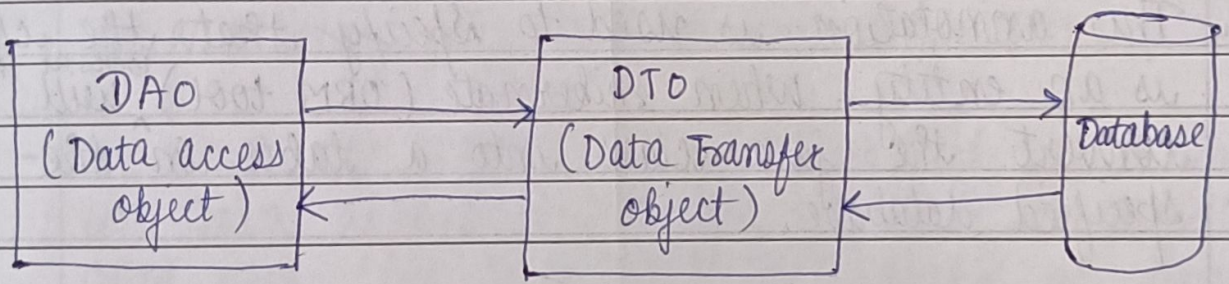
iv) @Column :-

- This annotation is used to control and modify the properties of a column.

- Ex:- i) To change the name of the Column :-
@Column (name = " value)

- ii) To add unique constraint :-
@ Column (Unique attribute = true / false)

Layers in Hibernate :-



(dto)

- DTO layer will consist of the class that is supposed to be the table in the database (Entity class). As the object of this class is transfered as data between java app^n and database. Hence, the name Data Transfer Object.

(dao)

- DAO layer will consist of the classes where operation on DTO is accessed. Hence, the name Data Access Object.

Date : 13/04/24

- Controller layer consist of class which has main method in it.

JPQL Input Parameters (used only in where or having clause)

There are 2 ways you can pass a value for a JPQL query :

i) JPQL positional parameters
ii) JPQL named parameters

i) JPQL positional parameters :-

Rules to be followed :-

i) Input parameters are designated by "?" prefix followed by an integer.
Ex: ?1

ii) Input parameters are numbered starting from 1.

ii) JPQL named parameters :-

- A named parameter is an identifier ie. prefixed with ":".
- Named parameters are case-sensitive.
- Ex:- : email

Date : 15/04/24

```
SELECT u
FROM User u
WHERE u.email = ?1
```
} Positional parameter

```
SELECT u
FROM User U
WHERE u.email = :email
```
} Named parameter

Hibernate Mapping

* Hibernate mapping :-

- Hibernate mapping is one of the key features provided by Hibernate which is used to establish the relationship between two database tables.

- To achieve hibernate mapping, hibernate provides 4 different ways :-
  i) OneToOne
  ii) OneToMany
  iii) ManyToOne
  iv) ManyToMany

• Note :-

- If the relationship is defined in only one of the tables, it is known as uni-directional mapping.

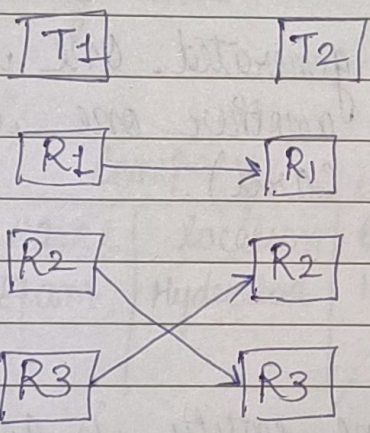
- If the relationship is defined in both the tables, it is known as bi-directional mapping.

Date :

i) OneToOne_uni :-

- When one record from table1 is related to exactly one record from table2. It is called as OneToOne mapping.

- Ex:- Person & Aadhar card , Employee & Salary Account
  Student & Id
  Person & PAN




```
@Entity                    @Entity
Person                     Aadhar_Card
  @Id                        @Id
  - id                       - id
  - name                     - name
  - phone                    - address
  - address
  @OneToOne
  AadharCard ac;
```

// Owning Side (dependent)

// Non-Owning Side (Independent)

PK Person — FK → PK aadhar card

id	name	phone	address	a-id
1	Sakshi	5446218373	Pune	101

Child Table

id	name	address
101	Sakshi	Pune

Parent Table

Violates the SQL & EF Codd rule & Primary Key

Date : 16/04/24

* How to achieve OneToOne mapping ?

- Here we will create a reference variable of one class ~~or~~ inside another class that will be ~~annoted~~ annotated as @OneToOne.

- By considering this annotation, Hibernate will create a foreign key inside owning side.

- Here, two tables will be generated. One is for owning side (Person) & another one is for non-owning side (Aadhar Card).

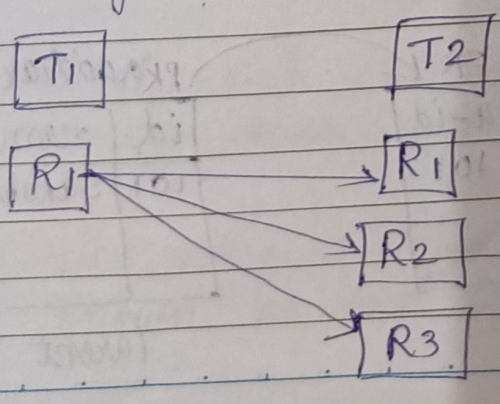
ii) OneToMany - uni :-

- In OneToMany mapping, one entity is associated with many instances of other entity.

OR

When one record from table 1 is related to more than one record from table 2, then it is known as oneToMany mapping.

- Ex :- ✓ Company & Employees
School & students
Country & States
Person & bankaccounts
State & districts



Date :

```
@Entity
Company
  @Id
  - id
  - name
  - location
  @OneToMany (For F.K)
  List<Employee> list;

// Owning side
```

```
@Entity
Employee
  @Id (P.K)
  - id
  - name
  - phone
  - address

// Non-Owning side
```

PK — Company — FK

id	name	location	e_id
101	Epam	Hyderabad	1, 2, 3

PK — Employee

id	name	phone	address
1	Sakshi	88888888	Pune
2	Rutika	999999	Mumbai
3	Pallavi	143491	Delhi

Company_Employee

FK	c_id	e_id	FK
	101	1	
	101	2	
	101	3	

- By considering the scenarios of company & employee, one company can have multiple employees.
- To achieve this we have to create employee object inside company class with the help of list
  ie. List<Employee> list;
  It must be annoted with @OneToMany

Date :

- In OneToMany mapping, 3 tables will be generated in the database. One is for owning side (Company), one is for non-owinging side (Employee) and one for storing foreign keys (Intermediate table) to achieve relationship.

- This is done because, if two tables are created, it violates E.F Codd's first rule (atomicity) and we have to make primary key of company as duplicate. It is ~~no~~ not possible. Hence, primary keys of both the tables are stored in seperate tables as foreign keys.

Note :-

- If the flow of relationship is towards many, we have to create a collection of objects to achieve Has-a-relationship.
In this case, Hibernate will create an intermediate table to define the relationship.

For :- Save → Entity object
delete }
Select } → PK
update → PK, Entity object

Date : 17/04/24

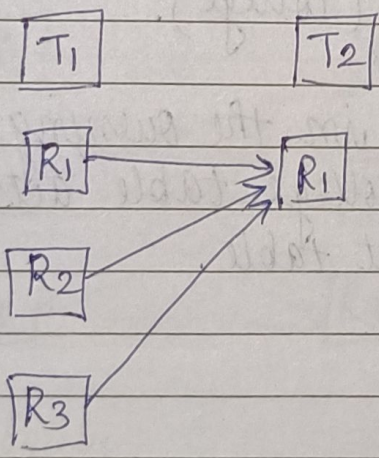
iii) ManyToOne_uni :-

- In ManyToOne mapping many instances of one entity is associated with one instance of other entity.

OR

When more than one records from table 1 are related to exactly one record from table 2, it is known as ManyToOne mapping.

- Ex :- Students & College , BankAccount & Person
  Employees & Company , ~~Hospital~~
  Branches & Hospital




```
@Entity
Student
  @Id
   -id
   - name
   - phone
   - address
  @ManyToOne
    College c;

// Owning side
```

```
@Entity
College
  @Id
   -id
   - name
   - fees

// Non - owning side
```

Date :

P.K Student

id	name	phone	address	C_id
1	Sakshi	8888888	Pune	101
2	Rutika	999999	Mumbai	101
3	Abhi	777777	Gujrat	101

F.K

P.K College

id	name	fees
101	FE	1L

- ManyToOne mapping means many objects are pointing towards one object.

- In this scenario, we are going to create a reference variable of college class inside student class.

- Here, two tables will be generated in the database. One for the owning side (Student) and another one is for non-owning side (College).

- The foreign key is present in the owning side. ie. the primary key of college table acts as foreign key in Student table.

iv) ManyToMany_Uni :-

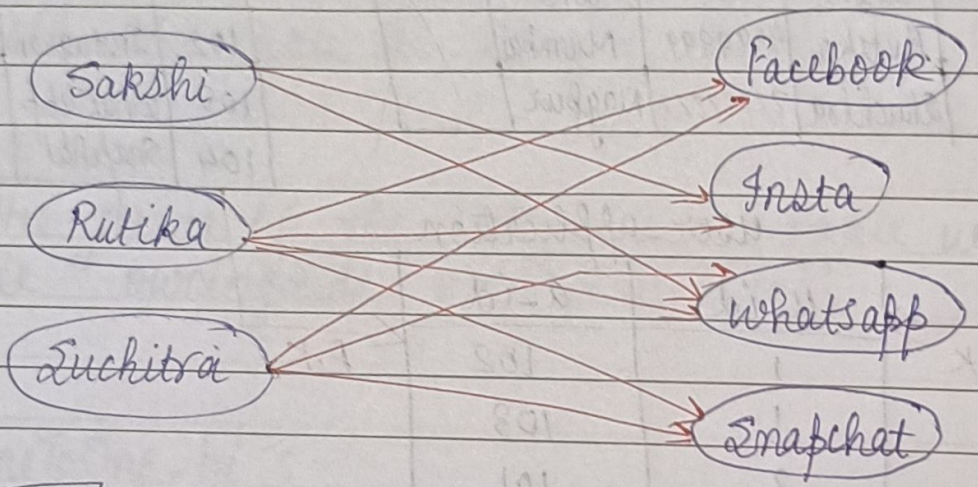
- When more than one records from table 1 are related to more than one records in table 2, it is known as ManyToMany mapping.

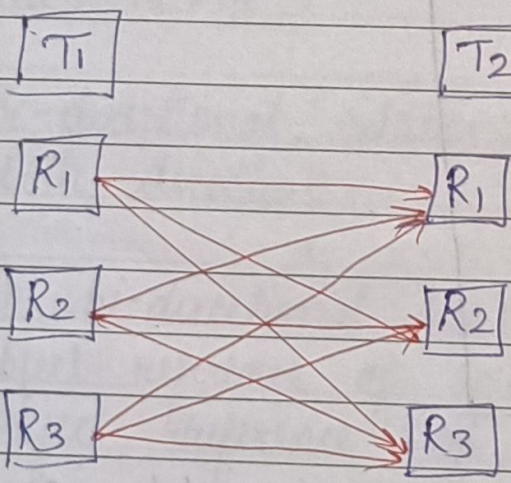
OR

In ManyToMany mapping, many instances of one entity is associated with many instances of other entity.
In simple words, many rows of one table is associated with another table.

Date : 18/04/24

Ex :- i) Many Students are associated with many Courses.
ii) Many Persons are associated with many languages
iii) Many Users are associated with many application.






```
@Entity
User
  @Id
  - id
  - name
  - phone
  - address
  @ManyToMany
  List<Application> apps;

  // owning side
```

```
@Entity
Application
  @Id
  - id
  - name
  - company

  // Non-owning side
```

Date :

User

PK

User	id	name	phone	address	a_id
	1	Lakshi	88888888	Pune	102, 103
	2	Rutika	9999999	Mumbai	
	3	Shuchitra	77777777	Nagpur	

F.K

Application

PK

id	name	company
101	Facebook	–
102	Instagram	–
103	whatsapp	–
104	Snapchat	–

user_application

u_id	a_id
1	102
1	103
2	101
2	102
2	103
2	104
3	101
3	103
3	104

F.K

F.K

- ManyToMany represents a collection-value association.
- Here, any no. of entities can be associated with a collection of other entities.
- Consider two entity classes, User & Application. In this scenario, one user can have many applications as well as one application can have many users.
- In this case, we create list of one entity type inside another entity class. It should be annotated with @ManyToMany.

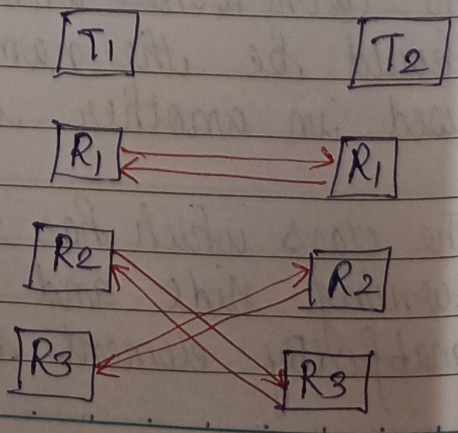
Date : 20/04/24

- In this mapping, 3 tables are generated in the database. 2(Two) for two entity classes & one table to store foreign keys.
- Here, third table is having primary keys of both the table as foreign keys.

- **Note :-**
- The name of the intermediate table will always be "owningSide_nonOwningSide".

v) OneToOne-bi :-

- Bi-directional relationship provides accessing in both direction.
- In bi-directional OneToOne, a class A has an object reference of another class B and class B has object reference of class A.
- The annotation @OneToOne is used in both the classes.
- Ex :- Person & Aadhar card
  Student & Id
  Person & PAN



Date :

```
@Entity
Person
 @Id
 @GeneratedValues(strategy=
   -id          GenerationType.IDENTITY)
   -name
   -phone
   -address
 @OneToOne
 AadharCard ac;

 // Owning side
```

```
@Entity
AadharCard
 @Id
 @GeneratedValues (strategy =
   -id          GenerationType.
                IDENTITY)
   - name
   - address
 @OneToOne
 Person p;

 // Owning side
```

PK	Person			FK
id	name	phone	address	a-id
1	Sakshi	88888888	Pune	101

PK	AadharCard		FK
id	name	address	p-id
101	Sakshu	~~2405152563~~ Mumbai	1

- Without mappedBy attribute, hibernate will generate foreign key column in both the tables. To avoid that mappedBy should be used.

- We must use mappedBy element with one of the @OneToOne annotation. The value of mappedBy should be the name of reference variable used in another class.

- The class which has mappedBy attribute becomes the non-owning side and the class which doesn't have mappedBy element becomes the owning side.

Date :

- In Owning side, we can use ~~attonation~~ annotation @JoinColumn. The purpose of this annotation is to specify a foreign key column.

```
@Entity
Person
@Id
@GeneratedValue(Strategy=
  -id   GenerationType.IDENTITY)
  -name
  -phone
  -address
@OneToOne
@JoinColumn
AadharCard ac;

// Owning side
```

```
@Entity
AadharCard
@Id
@GeneratedValue(Strategy =
  -id   GenerationType.
        IDENTITY)
  -name
  -address

@OneToOne(mappedBy =
        "ac")
Person p;

// non-Owning side
```

Person

PK id	name	phone	address	F.K ac_id
1	Lakshi	8888888888	Pune	101

AadharCard

PK id	name	address
101	Rutika	Mumbai

Date :

* Annotations Used :-

i) @GeneratedValue :-

- This annotation is used to make hibernate to automatically generate a value for the mentioned property.

- The value is generated based on some generation strategy. (provided by an Enum called javax.persistence.GenerationType). The available strategies are i) IDENTITY
  ii) AUTO
  iii) SEQUENCE
  iv) TABLE

  } Self-learning Exercise

ii) @JoinColumn :-

- It is used to specify a column for joining an entity association.

- This annotation indicates the enclosing entity is the owner of the relationship and the corresponding table as a foreign key column which refers to the table of the non-owning side

mapping annotation (OneToOne, OneToMany, ManyToOne, ManyToMany)

Date :

iii) @mappedBy element :-

- We use mappedBy element with any one of the mapping annotations (Except @ManyToOne)
- The value of mappedBy element should be the name of reference ~~item~~ variable used in other class's reference variable.
- The element mappedBy is needed to specify which side is the owning side. Without mappedBy, hibernate will generate foreign key columns in both the tables.
- The side which has mappedBy element becomes the non-owning side and the other side becomes the owning side.

vi) OneToMany / ManyToOne-bi :-

- Ex:- Person & BankAccounts
- In this relationship, we use annotations @OneToMany and @ManyToOne.
- The annotation @OneToMany is used on the side which has collection reference. The annotation @ManyToOne is used on the side which has Object single valued object reference.

Date :

```
@Entity
person
@Id @GeneratedValue(Strategy=GenerationType.IDENTITY)
-id
-name
-phone
-address
@OneToMany
List<BankAccount> account;

// Owning side
```

```
@Entity
BankAccount
@Id @GeneratedValue(GenerationType.IDENTITY)
-id
-name
-ifsc
-balance

@ManyToOne
Person p;

// Owning side
```

PK — Person — F.K

id	name	phone	address
101	Sakshi	99999999	Pune

PK — BankAccount

id	name	ifsc	balance	p-id
1	SBI	SBI123	30000	101
2	HDFC	HDFC12	45000	101
3	BOI	BOI12	85000	101
4	ICICI	ICICI1	90000	101

F.K

Person_BankAccount

p_id	b_id
101	1
101	2
101	3
101	4

F.K

Date :

- We use mappedBy element to specify which side is the owning side.

- We use mappedBy with @OneToMany annotation.

Why? → In OneToMany unidirection, there will be 3 tables created for 2 entity classes wheras in ManyToOne uni-direction, 2 tables will be created for 2 entity classes.

```
@Entity
Person
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
-id
-name
-phone
-address
@OneToMany(mappedBy = "p")
List<BankAccount> account;

// Non-Owning Side
```

```
@Entity
BankAccount
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
-id
-name
-ifsc
-balance
@ManyToOne
@JoinColumn
Person p;

// Owning Side
```

Person

PK id	name	phone	address
101	Sakshi	88888888	Pune

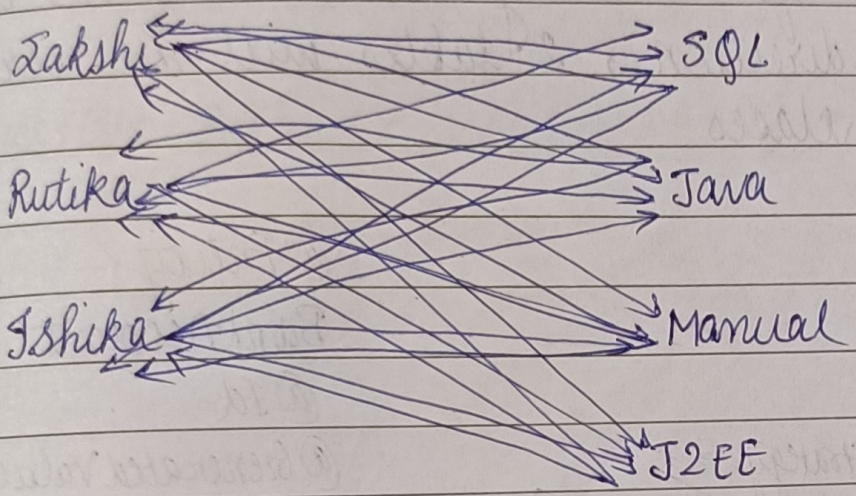
BankAccount

PK id	name	ifsc	balance	FK pid
1	SBI	SBI123	30000	101
2	HDFC	HDFC12	45000	101
3	BOI	BOI12	85000	101
4	ICICI	ICICI1	90000	101

Date : 22/04/24

ManyToMany - bi

In ManyToMany bi-direction, we are going to create list of objects of one Entity inside another Entity. It should be annotated with @ManyToMany




```
@Entity
Student
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
- sid
- name
- phone
- address
@ManyToMany
List<Cource> course;

// Owning Side
```

```
@Entity
Course
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
- cid
- name
- fees
- duration
@ManyToMany
List<Student> student;

//Owning Side
```

Date :

Course – Student

FK	Cid	s-id	FK
	101	1	
	101	2	
	101	3	
	102	1	
	102	2	
	102	3	
	103	2	
	103	3	
	104	1	
	104	3	

PK

Student

Sid	name	phone	address
1	Lakshi	432	Pune
2	Rutika	834	Mumbai
3	Ishika	967	Hydrabad

PK

Course

cid	name	fees	duration
101	SQL	8K	1.5
102	Java	15K	4
103	Manual	12K	1.5
104	J2EE	18K	2.5

FK

Sid	C.id
1	101
1	102
1	104
2	101
2	102
2	103
3	101
3	102
3	103
3	104

FK

- Here, In Bidirection we can fetch the data of one Entity with the help of another Entity we can access in both the directions

- Here, In ManyToMany Bi-direction four (4) Tables will be generated in the database.

* ManyToMany Bidirectional redundancy

- If ManyToMany bidirectional mapping is established between two entities, hibernate will create Intermediate table for both the Entities seperately.

- As @ManyToMany annotation is declared in both the entities, Hibernate is responsible to generate one Intermediate table for each of the annotation.

- This question leads to creation of one extra table which contains redundant data (duplicate data)

- To avoid the redundant table, we need to declare "mappedBy" attribute in one of the Entity class.

- With the help of this attribute, we can inform the hibernate that the mapping for this property we has been done by some other property.
  Hence, the intermediate table for this property should not be created.

- The mappedBy attribute is added in @ManyToMany which accepts the value in string format.
This string should indicate the property name from the related entity responsible for mapping.

```
@Entity
Student
@Id
@GeneratedValue(strategy=
  GenerationType.IDENTITY)
  - sid
  - name
  - phone
  - address
@ManyToMany
@JoinTable(joinColumns =
  @JoinColumn(name="sid"),
  inverseJoinColumns =
  @JoinColumn(name="cid"))

  - List<Course> courses;

// Owning side
```

```
@Entity
Cource
@Id
@GeneratedValue(strategy=
  GenerationType.IDENTITY)
  - cid
  - name
  - fees
  - duration
@ManyToMany(mappedBy=
  "Courses")

  - List<Student> student;

// Non-Owning
  side
```

Date :

PK

Student

sid	name	phone	address
1	Sakshi	483128	Pune
2	Rutika	99999	Mumbai
3	Ishika	443219	Hyderabad

PK

Course

Cid	name	fees	duration
101	SQL	8K	1.5
102	JAVA	15K	4
103	Manual	12K	1.5
104	J2EE	18K	2.5

Student_Course

F.K

F.K

Sid	C_id
1	101
1	102
1	104
2	101
2	102
2	103
3	101
3	102
3	103
3	104

- In the owning side of the association, we will use @JoinTable annotation which specifies the mapping of associations.

OTO & OTM & Cascading ✓

Date : 24/04/24

- @JoinTable :-

- This annotation is used to control and modify the properties of intermediate Table.

Format of @JoinTable annotation :-

@JoinTable(joinColumns = @JoinColumn(name = "PK of owning side"), inverseJoinColumns = @JoinColumn(name = "PK of non-owning side"))

Cascading

Imp * Cascading :-

- Cascading is a feature in hibernate, which is used to manage the state of mapped Entities whenever this state of its relationship owner is affected.

- When the relationship owner is saved / deleted, then the mapped entity associated with it should also be saved / deleted automatically.

- In other words, when we performed some action on the target entity, the same action will be applied to the associated entity.

- Cascading is done only at the owning side.

Date :

- All JPA-specific cascade operations are represented by javax.persistence.CascadeType enum containing entries like :-
  - PERSIST
  - MERGE
  - REMOVE
  - REFRESH
  - DETACH
  - ALL

* FETCH Types :-

- Fetch type defines how hibernate will fetch the data.
- It defines wheather hibernate will load data eagerly or lazily.
- We will use an enum called javax.persistence.FetchType
- It supports 2 types of loading
  i) EAGER
  ii) LAZY

i) EAGER :-

- When hibernate fetch the data from the owning side, both owning side and non-owning side will get fetched.

Date :

ii) LAZY :-

- When hibernate fetch the data from the owning side, only owning side data will be fetched.

Mappings	Default FetchType
OneToOne	EAGER
OneToMany	LAZY
ManyToOne	EAGER
ManyToMany	LAZY

- To modify the ~~fetch~~ FetchType, @MappingAnnotation(fetch = FetchType.EAGER / LAZY)

* Caching (used when we have to fetch the same data multiple times)

- Caching is a mechanism used to enhance the performance of an application. It is a buffer memory that lies between the application and the database.

- Cache memory stores recently used data items in order to reduce the number of database its as possible.

Date : _____

- There are mainly 2 types of caching :-
  i) First level caching
  ii) Second level caching

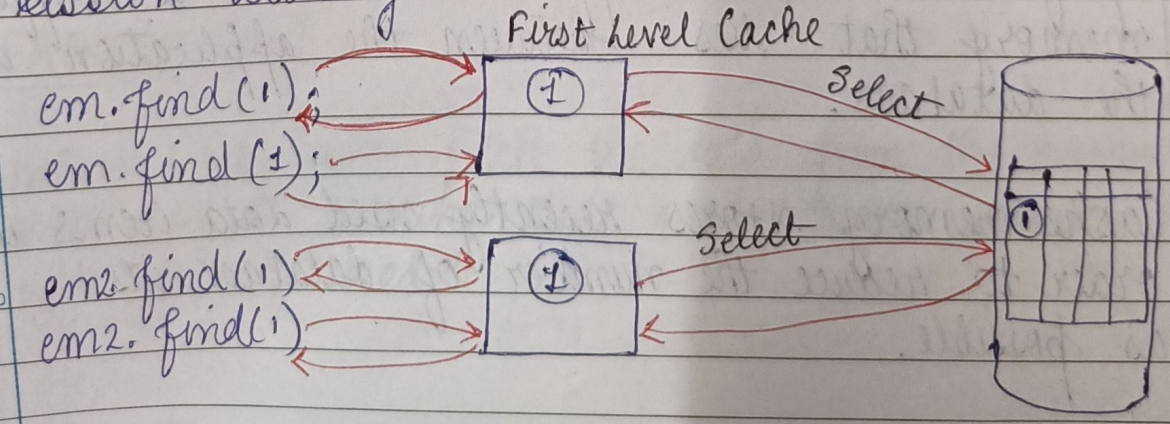
i) First level Caching (Implicitly provided) (session object holds the cache data)

- This is a mandatory cache which is present by default in hibernate.

- All the request objects passes through this cache. This cache can be utilize by application by sending many session objects. All the cache objects will be stored until one session is open.

- Database tries to minimize the number of hits, in case there are many statements commanded using session cache.

- Once the session is ended, this cache is also cleared and the objects it is holding are persisted, commited or disappeared without any updating, depending upon the time of session closing.



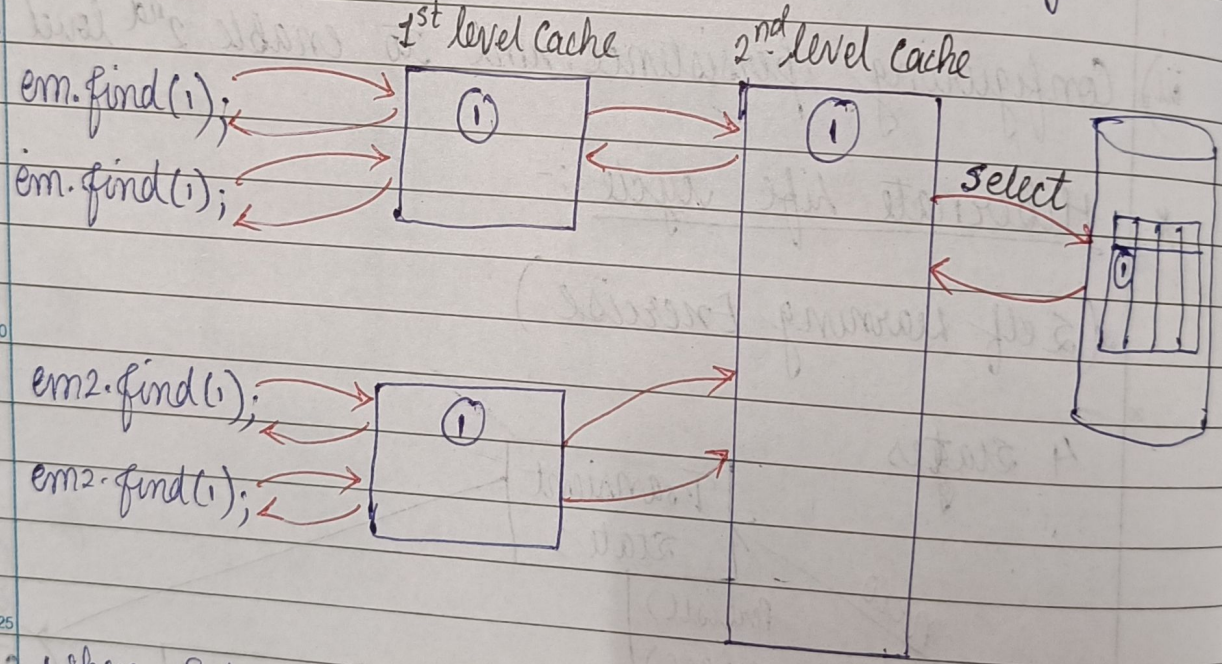
Date :

ii) Second level Caching (Explicitly provided) (Session_factory holds the cache data)

- Second level Caching is not enabled by default in hibernate. You have to explicitly enable it.

- The second level cache is accessible by the entire application. SessionFactory holds the data which can be accessed to all the sessions.

- Once the SessionFactory is closed, all the cache associated with it that is also removed from the memory.



- When hibernate session try to load an entity, it will first go to 1st level cache. If it didn't find, then it will look into the 2nd level cache and return the response (if available), but before returning the response it will store that object/data into the 1st level also. So, next time no need to come at the 2nd level.

Garbage Collector is the one of the reason why java is faster

Date :

When data is not found in the 2nd level cache, then it will go to the database to fetch the data. Before returning the response to the user, it will store object / data into both levels of cache, so next time it will be available at cache stages only.

* 3 steps to enable 2nd level cache :-

i) Add hibernate - ehCache dependancy in pom.xml

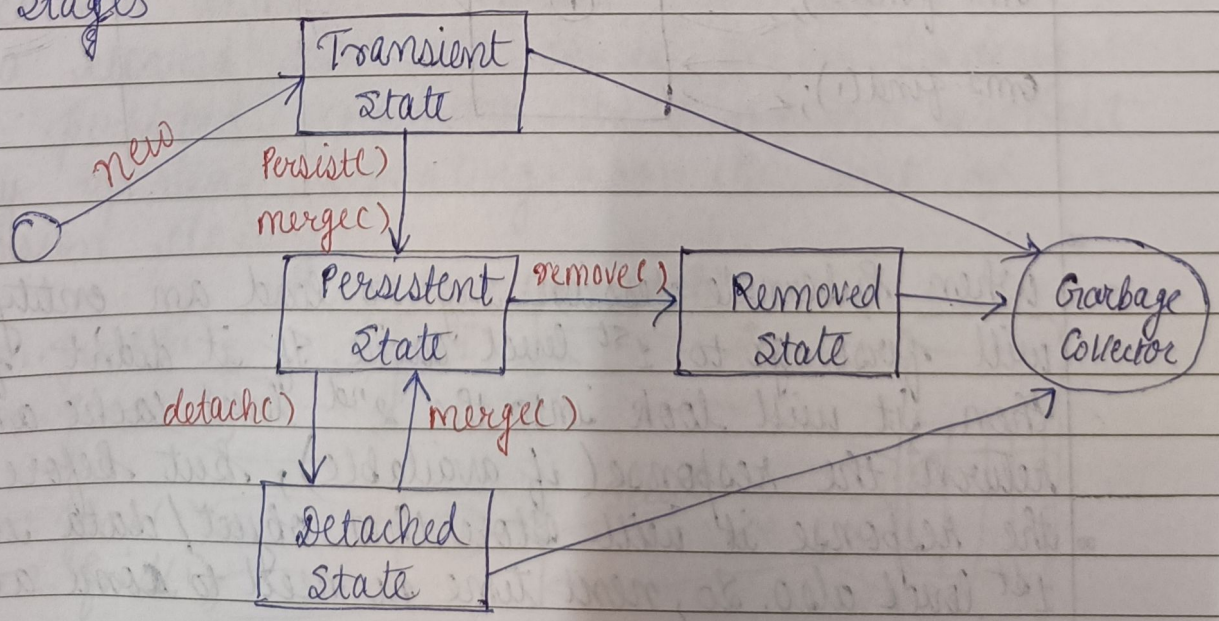
ii) Add @Cachable annotation to entity class

iii) Configuring persistence.xml to enable 2nd level cache.

* Hibernate life cycle :-

(Self learning Exercise)

4 stages



	No of Tables		NOS	OS	Cascade	FetchType (Default)
	uni	bi				
OTO	2	2 → 2	mappedBy	@JoinColumn	✓	Eager
OTM	3	3 → 2	mappedBy (OTM)	@JoinColumn (MTO)	✓	Lazy
MTO	2				X	Eager
MTM	3	4 → 3	mappedBy	@JoinTable(joinColumns = @JoinColumn(name = "PK of OS"), inverseJoinColumns = @JoinColumn(name = "PK of NOS"))	X	Lazy

Date :