DAY 5 JPQL
===

JPQL:
=====

similarities bt JPQL and sql:-

--keywords in the both the langauges are case insensetive

--GROUP BY,ORDER BY,where clause r similar

--aggregrate function r similar

--the way we express the condition to perform the CRUD operation is almost simmilar.

diff bt JPQL and sql:-

--sql queries are expressed in the term of table and columns, where as jpql query is expressed in the term of Entity class names and its variables.

--the name of the class and its variables are case sensitive.

--sql is not portable across multiple dbms, where jpql is portable.

```
sql> select name,marks from student; (name and marks are the column name
and student is the table name)

jpql> select name,marks from Student; (here name and marks are the
variables defined inside the Student class)
```

Note: we should not use * in jpql:
ex:

```
sql>select * from student;

jpql>from Student;   //projecting all the column
or
jpql>select s from Student s;

jpql> select s.name,s.marks from Student s;
```

steps to use the jpql in JPA application:-
----

step 1:- develop the JPQL query as string.

```
step 2:- create javax.persistnce.Query(I) object by calling
"createQuery(-)" method on the EM object.

ex:-

Query q =em.createQuery("JPQL query");


Query object the Object Oriented representation of JPQL.

step 3:- bind the values if any placeholders are used.(here we have 2
types of place holders 1.positional 2.named placeholders).

step 4:- submit the jpql query by calling either one of the following
methods:-

for select statments:-

List getResultList();   (if more than one record.)
Object getSingleResult();  (if atmost one record)


for insert/update/delete :-

int executeUpdate();  //this method should be called inside the tx area.

ex:-




in sql :-

select * from account;

in jpql:-

select a from Account a;

from Account;  //it is a shortcut


ex:- getting all the records from the DB:-
-----------------------------------------------

JPQLMain.java:-
-------------------


public class JPQLMain {

      public static void main(String[] args) {

            EntityManager em= EMUtil.provideEntityManager();


            //String jpql= "select a from Account a";
            String jpql= "from Account";
            Query q= em.createQuery(jpql);
            List<Account> list= q.getResultList();

            for(Account acc:list){
                  System.out.println(acc);
            }
      }
}
```

search on non-pk:-

```
      EntityManager em= EMUtil.provideEntityManager();


            //String jpql= "select a from Account a where a.name='Ram' ";
            String jpql= "from Account where name='Ram'";
            Query q= em.createQuery(jpql);

            List<Account> list= q.getResultList();

            for(Account acc:list){
                  System.out.println(acc);
            }
      }
```

if we conform that only one row will come then :-

```
      EntityManager em= EMUtil.provideEntityManager();


            //String jpql= "select a from Account a where a.name='Ram' ";
            String jpql= "from Account where name='Ram'";
            Query q= em.createQuery(jpql);

            //Object obj= q.getSingleResult();
            //Account acc= (Account)obj;

            Account acc= (Account)q.getSingleResult();

            System.out.println(acc);
```

--if the above query will return more that one result then it will throw a runtime exception

in order to avoid the downcasting problem we should use TypedQuery instead of Query obj.

--TypedQuery is the child interface of Query interface.

ex:-

```
            EntityManager em= EMUtil.provideEntityManager();
            //String jpql= "select a from Account a where a.name='Ram' ";
            String jpql= "from Account where name='Ram'";
            TypedQuery<Account> q= em.createQuery(jpql,Account.class);

            Account acc= q.getSingleResult();

            System.out.println(acc);
```

bulk update:-

```
            EntityManager em= EMUtil.provideEntityManager();

            String jpql= "update Account set balance=balance+500";

            Query q= em.createQuery(jpql);

            em.getTransaction().begin();
            int x= q.executeUpdate();
            em.getTransaction().commit();

            System.out.println(x+" row updated...");
```

using positional parameter:-

```
            EntityManager em= EMUtil.provideEntityManager();


            String jpql= "update Account set balance=balance+?5 where
name=?6";

            Query q= em.createQuery(jpql);

            q.setParameter(5, 1000);
            q.setParameter(6, "Manoj");

            em.getTransaction().begin();
            int x=q.executeUpdate();
            em.getTransaction().commit();


            System.out.println(x+" record updated...");
```

--index value can start with any number...

using named parameter:-

```
            EntityManager em= EMUtil.provideEntityManager();
            String jpql= "update Account set balance=balance+:bal where
name=:nm";

            Query q= em.createQuery(jpql);

            q.setParameter("bal", 600);
            q.setParameter("nm", "Ram");

            em.getTransaction().begin();
            int x=q.executeUpdate();
            em.getTransaction().commit();


            System.out.println(x+" record updated...");
```

1.--if we try to accees only one column then the return type will be :-

either String obj,
or any Wrapper class obj (Integer,Float)
or
LocalDate

2.--if all column then the return type will be the Entity class.(internally it will be mapped.)

3.if few columns then the return type will be Object[]. in this array each index will represent each column

name : String
balance: Integer
all columns : Account object

name,balance : Object[]

ex:- for 1 row and 1 column:-

```
      EntityManager em= EMUtil.provideEntityManager();

      Query q= em.createQuery(jpql);

       q.setParameter("ano", 105);

       String n= (String)q.getSingleResult();

       System.out.println(n);


//          TypedQuery<String> q=em.createQuery(jpql,String.class);
//
```

```
//          q.setParameter("ano", 105);
//
//          String n= q.getSingleResult();
//
//
//          System.out.println(n);
```

ex: multiple row and 1 column:-

```
            EntityManager em= EMUtil.provideEntityManager();


            String jpql= "select balance from Account";

            Query<Integer> q=em.createQuery(jpql);

            List<Integer> list= q.getResultList();

            System.out.println(list);
```

ex3:- few column and all rows:-

```
      EntityManager em= EMUtil.provideEntityManager();


            String jpql= "select name,balance from Account";

            Query q= em.createQuery(jpql);



            List<Object[]> results= q.getResultList();

            for(Object[] or: results) {

                   String name= (String)or[0];
                   int balance= (Integer)or[1];

                   System.out.println("Name is "+name);
                   System.out.println("Balance is :"+balance);

                  System.out.println("===========================");
            }

      }
```

few column with single record:

Demo.java:

```
package com.masai.usecases;

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.Query;
import javax.persistence.TypedQuery;

import com.masai.model.Account;
import com.masai.utility.EMUtil;

public class JPQLUseCase {

      public static void main(String[] args) {


            EntityManager em= EMUtil.provideEntityManager();

            String jpql= "select name,balance from Account where accno=
:ano";



//          Query q= em.createQuery(jpql);
//
//          q.setParameter("ano", 104);
//
//           Object obj= q.getSingleResult();
//
//                Object[] or= (Object[])obj;
//

            TypedQuery<Object[]> tq= em.createQuery(jpql, Object[].class);

            tq.setParameter("ano",104);

            Object[] or= tq.getSingleResult();




                  String name= (String)or[0];
                  int balance= (Integer)or[1];

                  System.out.println("Name is "+name);
                  System.out.println("Balance is :"+balance);

            em.close();
      }

}
```

aggregrate function:-

--any aggregrate function will return :-

min,max, count: Integer
avg : Double
sum : Long

ex:-

```
            EntityManager em= EMUtil.provideEntityManager();

            String jpql= "select sum(balance) from Account";

            TypedQuery<Long> q=em.createQuery(jpql,Long.class);

            long result= q.getSingleResult();

            System.out.println(result);
```

Named Queries:-
============

--if we require to write same query again and again in multiple Data access layer classes, it is recomended to use NamedQuery,

--in which we centralize the query with a unique name inside the Entity class.

and refer that name in all the Data access layer classes.

ex:-

Account.java:- (Entity class):-

```
@Entity
@NamedQuery(name = "account.getBalance",query = "from Account where
balance <:bal")
public class Account  {

      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      private int accno;
      private String name;
      private int balance;
```

JPQLMain.java:-

```
public class JPQLMain {

      public static void main(String[] args) {

            EntityManager em= EMUtil.provideEntityManager();


            Query q= em.createNamedQuery("account.getBalance");

            q.setParameter("bal", 5000);

            List<Account> list= q.getResultList();

            list.forEach(a -> System.out.println(a));

      }

}
```

NativeQueries:-
============

--here we write the Query in the term of tables and their columns. (normal sql)

```
            EntityManager em= EMUtil.provideEntityManager();


            String nq="select * from account"; //here account is the table name

            Query q= em.createNativeQuery(nq, Account.class);

            List<Account> list= q.getResultList();

            list.forEach(a -> System.out.println(a));
```

NamedNativeQuery:-

Account.java:-

```
@Entity
@NamedNativeQuery(name="allAccount" ,query = "select * from account",resultClass=Account.class)
public class Account  {

      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      private int accno;
      private String name;
      private int balance;
--
--
}
```

JPQLMain.java:-

```
public class JPQLMain {

	public static void main(String[] args) {

		EntityManager em= EMUtil.provideEntityManager();

		Query q= em.createNamedQuery("allAccount");

		List<Account> list= q.getResultList();

		list.forEach(a -> System.out.println(a));
	}

}
```

--Native queries are not recomended to use in realtime application developement.

Mismatched bt Object Oriented Representation and relational representaion of data:-
----

1.granularity mismatch :- HAS-A relationship problem

2.inheritence mismatch :- IS-A relationship problem

3.Association Mismatch :- table relationship problem

1.granularity mismatch :- HAS-A relationship problem:-
==

```
@Entity
class Employee{  --corse grain

@Id
int eid;
String ename;
int salary

Address addr;
}



//this type of class is known as value class or normal class, it is not
an Entity class
class Address{  --fine grain

String city;
String country;
String pincode;

}
```

an Entity can exist independently.

--at table level we don't have Has-A relationship. (it is Has-A relationship mismatch)

DAY 6 Mismatches
==

Mismatched bt Object Oriented Representation and relational representaion of data:-
----

1.granularity mismatch :- HAS-A relationship problem

2.inheritence mismatch :- IS-A relationship problem

3.Association Mismatch :- table relationship problem

1.granularity mismatch :- HAS-A relationship problem:-
==

```
@Entity
class Employee{  --corse grain

@Id
int eid;
String ename;
int salary

Address addr;

}
```

```
//this type of class is known as value class or normal class, it is not
an Entity class
class Address{  --fine grain

String city;
String country;
String pincode;

}
```

an Entity can exist independently.

--at table level we don't have Has-A relationship. (it is Has-A relationship mismatch)

solution for the above HAS-A relation problem:-
----

approach 1:-

--we need to create a single table with all column (all for corse grain + all for fine grain classes)

apply @Embeddable at the top of Address class or @Embedded at the top of Address addr varible indside the Employee Entity.

ex:-

Address.java:-

```
public class Address {

      private String state;
      private String city;
      private String pincode;
--
--
}
```

Employee.java:-

```
@Entity
public class Employee {

      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int eid;
      private String ename;
      private int salary;

      @Embedded
      private Address addr;  //here Address obj will be treated as value
obj
--
--
--

}
```

Demo.java:-

```
public class Demo {

      public static void main(String[] args) {

            EntityManager em= EMUtil.provideEntityManager();

            Employee emp=new Employee();
            emp.setEname("Ram");
            emp.setSalary(7800);
            emp.setAddr(new Address("Maharastra", "pune", "75455"));


            //Address adr=new Address("maharastra", "pune","75455");
            //emp.setAddr(adr);

            em.getTransaction().begin();

            em.persist(emp);

            em.getTransaction().commit();

            System.out.println("done...");

      }

}
```

--if we try to take 2 address (one for home and another for office) and then try to persist the employee obj we will get exception "repeated column"

--we can solve this problem by overriding the column names of Embedded obj by using "@AttributeOverrides" annotation.

ex 2:-
=====

Employee.java:-

```
@Entity
public class Employee {
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	private int eid;
	private String ename;
	private int salary;

	@Embedded
	@AttributeOverrides({


	@AttributeOverride(name="state",column=@Column(name="HOME_STATE")),

	@AttributeOverride(name="city",column=@Column(name="HOME_CITY")),

	@AttributeOverride(name="pincode",column=@Column(name="HOME_PINCODE
"))

	})
	private Address homeAddr;

	@Embedded
	@AttributeOverrides({


	@AttributeOverride(name="state",column=@Column(name="OFFICE_STATE")
),

	@AttributeOverride(name="city",column=@Column(name="OFFICE_CITY")),

	@AttributeOverride(name="pincode",column=@Column(name="OFFICE_PINCO
DE"))

	})
	private Address officeAddr;

--
--
--

}

Demo.java:-
---------------


public class Demo {

	public static void main(String[] args) {

		EntityManager em= EMUtil.provideEntityManager();

		Employee emp=new Employee();
		emp.setEname("Ram");
		emp.setSalary(7800);
		emp.setHomeAddr(new Address("Maharastra", "pune", "75455"));
		emp.setOfficeAddr(new Address("Telengana","hydrabad",
"785422"));


		em.getTransaction().begin();
```

```
            em.persist(emp);

            em.getTransaction().commit();

            System.out.println("done...");

      }

}
```

approach 2:-
----

if any emp has more than two address then taking too many columns inside a table will violates the rules of normalization.

--to solve this problem we need to use @ElementCollection annotaion, and let the user add the multiple addresses using List or Set.

--in this case ORM s/w will generate a seperate table to maintain all the addresses details with a Foreign key that reffers the PK of Employee table.

ex:-

Employee.java:-

```
      @Entity
      public class Employee {

      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int eid;
      private String ename;
      private int salary;

      @ElementCollection
      @Embedded
      private Set<Address> addresses=new HashSet<Address>();

      //
      }
```

Note: it is recomened to override equals() and hashCode() method if we want to put any user-defined objects inside the HashSet or a key of a HashMap.

Address.java:

```
package com.masai.model;

import java.util.Objects;
import javax.persistence.Embeddable;


public class Address {


	private String state;
	private String city;
	private String pincode;
	private String type;



	@Override
	public int hashCode() {
		return Objects.hash(city, pincode, state, type);
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Address other = (Address) obj;
		return Objects.equals(city, other.city) && 
Objects.equals(pincode, other.pincode)
				&& Objects.equals(state, other.state) && 
Objects.equals(type, other.type);
	}
	public String getState() {
		return state;
	}
	public void setState(String state) {
		this.state = state;
	}
	public String getCity() {
		return city;
	}
	public void setCity(String city) {
		this.city = city;
	}
	public String getPincode() {
		return pincode;
	}
	public void setPincode(String pincode) {
		this.pincode = pincode;
	}
	public String getType() {
		return type;
	}
	public void setType(String type) {
		this.type = type;
	}
	public Address(String state, String city, String pincode, String 
type) {
		super();
		this.state = state;
		this.city = city;
		this.pincode = pincode;
		this.type = type;
	}


	public Address() {
		// TODO Auto-generated constructor stub
	}
	@Override
	public String toString() {
		return "Address [state=" + state + ", city=" + city + ",
pincode=" + pincode + ", type=" + type + "]";
	}




}
```

```
Demo.java:-
--------------

public class Demo {

	public static void main(String[] args) {

		EntityManager em= EMUtil.provideEntityManager();

		Employee emp=new Employee();
		emp.setEname("Ram");
		emp.setSalary(7800);

		Employee emp= new Employee();
		emp.setEname("Ramesh");
		emp.setSalary(6800);
		emp.getAddresses().add(new Address("Mh", "Pune",
"787887","home"));
		emp.getAddresses().add(new Address("MP", "Indore",
"584542","office"));


		em.getTransaction().begin();

		em.persist(emp);

		em.getTransaction().commit();

		System.out.println("done...");

	}

}
```

--when we execute the above application 2 tables will be created :-

1.employee :- which will contains only Employee details (it will not contains any details of any address)

2.employee_addresses :- this table will contains the details of all the addresses with a FK column employee_eid which reffers the eid column of employee table.

Note:- if we want to change the 2nd table 'employee_addresses' and the FK column with our
our choice name then we need to use @JoinTable and @JoinColumn

ex:-

Employee.java:-

```
@Entity
public class Employee {

      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int eid;
      private String ename;
      private int salary;

      @ElementCollection
      @Embedded
      @JoinTable(name="empaddress",joinColumns=@JoinColumn(name="emp_id")
)
      private Set<Address> addresses=new HashSet<Address>();

      --
      --
      }
```

with the above example the 2nd table will be created by the name "empaddess" and the
FK column will be by the name "emp_id".

example:

Demo.java:

```
package com.masai.model;

import java.util.List;
import java.util.Set;

import javax.persistence.EntityManager;
import javax.persistence.Query;

import com.masai.utility.EMUtil;
```

```
public class Demo {

      public static void main(String[] args) {


            EntityManager em= EMUtil.provideEntityManager();

            //get all the Address of a Employee whose name is Ramesh

            String jpql="from Employee where ename='Ramesh'";

            Query q= em.createQuery(jpql);

            List<Employee> emps= q.getResultList();

            for(Employee emp:emps) {

                  Set<Address> addrs= emp.getAddresses();

                  for(Address adr:addrs) {

                        System.out.println(adr);
                  }


            }

            em.close();

      }


}
```

eager and lazy loading:-

--by default ORM s/w (Hibernate) perform lazy loading while fetching the objs, when we fetch the parent obj(first level obj),then only the first level obj related data will be loaded into the memory,but the 2nd level obj related data will be loaded at time of calling the 2nd level object related methods.

ex:-

Demo1.java:-

```
public class Demo {

      public static void main(String[] args) {

            EntityManager em= EMUtil.provideEntityManager();
            Employee emp= em.find(Employee.class, 10);

            em.close();  // even though before closing the EM obj we got 
the Employee obj
            //here only Employee related obj will be loaded ,address obj 
data will be not be loaded
                  //so while fetching the address related data we will get 
an exception

            System.out.println(emp.getEid());
            System.out.println(emp.getEname());
            System.out.println(emp.getSalary());

            System.out.println("All Address are:-");

            System.out.println("===========================");
            Set<Address> addreses= emp.getAddresses();

            for(Address ad:addreses){
                  System.out.println("city :"+ad.getCity());
                  System.out.println("state :"+ad.getState());
                  System.out.println("Pincode :"+ad.getPincode());

                  System.out.println("***************************");
            }

            System.out.println("done...");
      }

}
```

--to solve the above problem we need to use Eager loading:-

ex:-

```
Employee.java:-
------------------

@Entity
public class Employee {

      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int eid;
      private String ename;
      private int salary;

      @ElementCollection(fetch=FetchType.EAGER)
      @Embedded
      @JoinTable(name="empaddress",joinColumns=@JoinColumn(name="emp_id")
)
      private Set<Address> addresses=new HashSet<Address>();
--
--

}
```

Association Mismatch:- table relationship problem:-
===

--at the table level different types of tables will participate in different kind of relationships

ex:-

1.one to one (person ----- Driving licence) :- PK and FK(unique)

2.one to many (Dept ----Emp) :- PK and FK (i.e PK of Dept will be inside the Emp as FK)

3.many to many (student --- course) :- we need to take the help of 3rd table(linking table)

---to access the meaningfull information from the multiple tables we need to establish the relationship.

--these relationship enable us to navigate from one table record to another table records.

--to navigate from one table to another table,our tables must be in a relationship.

--when tables in the relationship then the Entity classes which represents the tables should also be in the relationships accordingly. so that objs of these classes should also be in a relationship .

-- so we can navigate from one obj details to another obj details.

--JPA supports the relationship bt the Entity classes not only with the cardinality but also with the
direction

--uni-directional and bi-directional is the another classification of relationship.

---in uni-direc, we can define child Entity obj inside the parent Entity , or parent Entity reff inside the
child Entity , but both are not possible.

--with this relation, we can access the child class obj from parent obj or parent class obj from the
child class obj, both not possible at a time.

--in bi-directional :- we define child Entity obj inside the parent Entity and parent Entity obj inside the
child Entity,(navigation is possible from the either one of the any obj)

so JPA supports 4 types of relationships:-

1.one to one

2.one to many
3.many to one
4.many to many (it is by defualt bi-directional only)

One-to-Many unidirectional:- (from Dept to Emp)
----

one Dept can have multiple Emp ,

step 1:- here we need to develop child Entity class first as individual.(Employee Entity)

step 2:- develop a parent Entity class with its own properties and declare one extra Collection type of Child
Entity class property (either List of child entity class or Set of child entity class).

and apply @OneToMany annotation to this property ex:-

```
      @OneToMany
      private List<Employee> emps=new ArrayList<Employee>();
```

Employee.java:-

```
      @Entity
      public class Employee {

      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int empId;
      private String name;
      private int salary;

      --
      --
      }
```

Department.java:-

```
      @Entity
      public class Department {

      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int deptId;
      private String dname;
      private String location;

      //it is the extra property to maintain OTM relationship
      @OneToMany
```

```
	private List<Employee> emps=new ArrayList<Employee>();
	--
	--
	}
```

Demo.java:-

```
public class Demo {

	public static void main(String[] args) {

	EntityManager em=EMUtil.provideEntityManager();


	Employee emp1=new Employee();
	emp1.setName("ram");
	emp1.setSalary(8500);

	Employee emp2=new Employee();
	emp2.setName("ramesh");
	emp2.setSalary(7500);

	Department dept=new Department();

	dept.setDname("HR");
	dept.setLocation("Kolkata");

	//associating both employee with a  dept obj
	dept.getEmps().add(emp1);
	dept.getEmps().add(emp2);



	em.getTransaction().begin();

	em.persist(emp1);
	em.persist(emp2);
	em.persist(dept);

	em.getTransaction().commit();

	em.close();

	System.out.println("done...");

	}
}
```

--with the above application, here for both the Entity classes 2 seperate tables will be created independently(they does not have info about each other.) , in addition to that one seperate linking table will be created which contains the PK of both the tables.

this seperate table name and its column names are:-

```
department_employee  :- table name
	department_deptid  :- it reffers deptid of department table
```

```
      emps_empid;  :- it will reffers empid of employee table
```

--in the above application we have saved first, all the child entity obj then we saved the parent entity obj.

--but if we want that once we persist the parent obj, automatically all the child object also should be persisted, then we need to use cascading option:-

ex:-

```
      @OneToMany(cascade= CascadeType.ALL)
      private List<Employee> emps=new ArrayList<Employee>();
```

--we can change the 3rd generated table name as well as their column names also :-

ex:-

```
      @OneToMany(cascade= CascadeType.ALL)
@JoinTable(name="dept_emp",joinColumns=@JoinColumn(name="did"),inverseJoi
nColumns=@JoinColumn(name="eid"))
      private List<Employee> emps=new ArrayList<Employee>();
```

here the 3rd table name will become :- dept_emp;

and both column names will be :-

did(which reffers the PK of department table) and eid(which reffers PK of employee table)

Note: - Department Entity class will take the help of this 3rd table to navigate the details of Employee Entity

ex:- adding another employee in the exsisting department-

Demo.java:-

```
public class Demo {

      public static void main(String[] args) {

      EntityManager em=EMUtil.provideEntityManager();


      Employee emp=new Employee();
      emp.setName("Amit");
      emp.setSalary(6500);
      Department dept= em.find(Department.class, 1);

      List<Employee> emps= dept.getEmps();

      em.getTransaction().begin();

      emps.add(emp);

      em.getTransaction().commit();


      System.out.println("done...");

      }
}
```

getting all the Employees from the Department "HR";
==

```
package com.masai.model;

import java.util.List;
import java.util.Set;

import javax.persistence.EntityManager;
import javax.persistence.Query;

import com.masai.utility.EMUtil;

public class Demo {

      public static void main(String[] args) {


            EntityManager em= EMUtil.provideEntityManager();


            String jpql= "select emps from Department where dname='HR'";

            Query q= em.createQuery(jpql);

            List<Employee> allemps= q.getResultList();


            System.out.println(allemps);


            System.out.println("done...");



      }


}
```

Note: in a single list we get all the Employee list.

DAY 07 Relations
==
=============

Many to one (uni-directional):-

--from Emp to Dept

--in one to many we navigate from parent to child, whereas in many to one we navigate from child to parent.

--MTO association means many obj of child Entity holds the single obj of parent Entity

--here we need to take a Department class reference variable inside the Employee class and apply the @ManytoOne annotation.

--and Department Entity class should not have any reff of Employee class, since it is a uni-direcitional.

ex:-

Department.java:-

```
@Entity
public class Department {

      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int deptId;
      private String dname;
      private String location;

      --
      --
      }
```

Employee.java:-

```
@Entity
public class Employee {

      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int empId;
      private String name;
      private int salary;
      @ManyToOne(cascade=CascadeType.ALL)
      private Department dept;
--

}
```

Demo.java:-

```
public class Demo {

      public static void main(String[] args) {

      EntityManager em=EMUtil.provideEntityManager();

      Department dept=new Department();
      dept.setDname("Sales");
      dept.setLocation("mumbai");

      Employee emp1=new Employee();
      emp1.setName("ram");
      emp1.setSalary(7800);
      emp1.setDept(dept);

      Employee emp2=new Employee();
      emp2.setName("ramesh");
      emp2.setSalary(8850);
      emp2.setDept(dept);


      em.getTransaction().begin();

      em.persist(emp1);
      em.persist(emp2);

      em.getTransaction().commit();

      System.out.println("done...");

      }
}
```

--here a seperate table will not be created, instead inside the child table one FK column will be created which will reffer the Department table PK.

--here employee table is the owner of the relationship,

bydefault name of this FK will be "dept_deptid" with respect to above application.

--if we want to change this name then we need to use

ex:-

```
	@ManyToOne(cascade=CascadeType.ALL)
	@JoinColumn(name="did")
	private Department dept;
```

getting the details of Department based on employee Id:-

```
	EntityManager em=EMUtil.provideEntityManager();



	Employee emp= em.find(Employee.class, 3);

	Department dept= emp.getDept();

	System.out.println(dept.getDeptId());
	System.out.println(dept.getDname());
	System.out.println(dept.getLocation());
```

One to Many (bidirectional):-

--here we need to combine above both approach , i.e inside Dept class we need take the List<emp> variable and inside Emp class we need to take the Dept class simple variable

--here we can apply cascading in both side.

***while persisting the objs we need to associate both objs with each ohter. otherwise we will not get the desired result.

ex:-

Employee.java:-

```
@Entity
public class Employee {

	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	private int empId;
	private String name;
	private int salary;

	@ManyToOne(cascade=CascadeType.ALL)
	private Department dept;
}
```

Department.java:-

```
-----------------------


@Entity
public class Department {

	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	private int deptId;
	private String dname;
	private String location;

	@OneToMany(cascade=CascadeType.ALL)
	private List<Employee> emps=new ArrayList<Employee>();
--
}


Demo.java:-
---------------


public class Demo {

	public static void main(String[] args) {

	EntityManager em=EMUtil.provideEntityManager();

	Department dept=new Department();
	dept.setDname("Marketing");
	dept.setLocation("Kolkata");

	Employee emp1=new Employee();
	emp1.setName("Sunil");
	emp1.setSalary(7800);
	emp1.setDept(dept); //associating dept with emp1

	Employee emp2=new Employee();
	emp2.setName("Suresh");
	emp2.setSalary(8800);
	emp2.setDept(dept); //associating dept with emp1

	//here both emp got the dept details..

	//now we need to give both emp details to the dept
	//associating both emp with the dept

	dept.getEmps().add(emp1);
	dept.getEmps().add(emp2);


	em.getTransaction().begin();

	em.persist(dept);

	em.getTransaction().commit();

	System.out.println("done...");

	}
```

}

--here one 3rd table will be created, by using this Dept Entity will get the details of Emp Entity.
and one FK column will be generated inside the emp table by using this Emp Entity get the details of Dept.

--in order to tell the ORM s/w while navigating from Dept to Emp,don't use the 3rd linking table , relationship is already maintained inside the employee table , so instead of using 3rd table use the employee table reff we use "mappedBy" property inside the @OneToMany annotation with the value:- the variable defined in another side.

Employee.java:-

```
@Entity
public class Employee {

      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int empId;
      private String name;
      private int salary;

      @ManyToOne(cascade=CascadeType.ALL)
      @JoinColumn(name="did")
      private Department dept; //this variable is used in mappedBy of
Department class

}
```

Department.java:-

```
      @Entity
      public class Department {

      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int deptId;
      private String dname;
      private String location;

      @OneToMany(mappedBy="dept" ,cascade=CascadeType.ALL)
      private List<Employee> emps=new ArrayList<Employee>();

--
}
```

Demo.java:-
----

same as above app

List out all the employees working in perticular dept:-

Demo.java:-

```
public class Demo {

	public static void main(String[] args) {

	EntityManager em=EMUtil.provideEntityManager();

	Department d= em.find(Department.class, 1);

	List<Employee> emps= d.getEmps();

	emps.forEach(e ->{

		System.out.println(e.getEmpId());
		System.out.println(e.getName());
		System.out.println(e.getSalary());

	});

	System.out.println("done...");

	}
}
```

ManytoMany:-

ManyTOMany :- (it is binature a bidirectional association)
===========

--it is a combination of one-to-many association from parent and one-to-many association from child

--at table level,to establish a many-to-many relationship we need a third linking table.

steps to achive the MTM relationship bt classes in HB:-
----

incase of MTM relationship we need to take both side collection properties and we need to apply @ManyToMany anno on the top of both side variables.

ex:-

```
public class Department
{

@ManyToMany(cascade = CascadeType.ALL)
List<Employee> empList = new ArrayList<>();
--
--

}



public class Employee
{
@ManyToMany(cascade = CascadeType.ALL)
List<Department> deptList = new ArrayList<>();

}
```

Note: while persisting the record we need to assoicate both objects with each other.

ex:-

```
            Department d1 = new Department();
            d1.setDname("sales");
            d1.setLocation("kolkata");


            Department d2 = new Department();
            d2.setDname("Marketing");
            d2.setLocation("delhi");




            //creating employee without department
            Employee emp1 = new Employee();
            emp1.setName("ram");
            emp1.setSalary(50000);


            Employee emp2 = new Employee();
            emp2.setName("dinesh");
            emp2.setSalary(30000);

            //associating department with both employees(ram,dinesh) with
dept sales
            emp1.getDeptList().add(d1);
            emp2.getDeptList().add(d1);
            //associating dept(sales) with both emp1 and emp2

            d1.getEmpList().add(emp1);
            d1.getEmpList().add(emp2);



            //assume dinesh is working in 2 dept(sales and marketing)
            emp2.getDeptList().add(d2);
            d2.getEmpList().add(emp2);



            em.getTransaction().begin();

            em.persist(d1);
            em.persist(d2);

            em.getTransaction().commit();

            System.out.println("done");
      }
```

--here if we save the both the objs by associating them together then it will create total 4 tables

```
department
employee
department_employee(Employee_empid, deptList_did)
employee_department(Department_did,empList_empid)
```

--in order to generate only one linking table then we need to use mappedBy property here also(in any side).

ex:-

```
@Entity
public class Department {

      @ManyToMany(cascade = CascadeType.ALL,mappedBy = "deptList")
      List<Employee> empList = new ArrayList<>();;

}
```

--here Employee obj doing the mapping not the Department obj.
so only one linking table will be created by name employee_department.

--here also we can mention the JoinTable name and joinColumn names,inverseColumn name ,this should be inside the Employee class.

ex:-

```
@Entity
public class Employee {


      @ManyToMany(cascade = CascadeType.ALL)
      @JoinTable(name = "emp_dept", joinColumns =
@JoinColumn(name="empid"),inverseJoinColumns = @JoinColumn(name="deptid")
)
      private List<Department> deptList;

}
```

Navigating from emp to dept:-

```
            List<Department> dlist =em.find(Employee.class,
2).getDeptList();

            System.out.println(dlist);
```

Navigating from dept to emp:-

```
            List<Employee> dlist =em.find(Department.class,
1).getEmpList();

            System.out.println(dlist);
```

OneToOne:-
=========

--at table level,we can maintain OTO relation by taking FK as Unique.

unidirectional:-

--Assume one department has only one employee and one employee belongs from only one dept.

we can take other example also

ex:-

Emp --> Address

Person --> DL

--here we need to use @OneToOne annotation

```
@Entity
public class Department {

     @OneToOne(cascade = CascadeType.ALL)
     private Employee emp;

}
```

Main class:-

```
           Department d1=new Department();
           d1.setDname("Sales");
           d1.setLocation("kolkata");



           Employee emp = new Employee();
           emp.setEname("Ram");
           emp.setSalary(8500);

           d1.setEmp(emp);

           em.getTransaction().begin();

           em.persist(d1);

           em.getTransaction().commit();
```

--here 2 table will be created

1.employee (empid,name,salary)

2.department(did,dname,dlocation, emp_empid) (this emp_empid will be the FK)

--if we want to change this auto generated FK column name then we need to apply @JoinColumn anno

ex:-

```
@OneToOne
@JoinColumn(name="eid")
private Employee emp;
```

bidirectional:-
----

onetoone bidirectional :-
----

here on both side define opposit class variables:-

ex:-

```
Department:-
      @OnetoOne
      private Employee emp

Employee:-
      @OneToOne
      private Department dept
```

ex:-

```
            Department d1=new Department();
            d1.setDname("Sales");
            d1.setLocation("kolkata");


            Employee emp = new Employee();
            emp.setEname("Ram");
            emp.setSalary(8500);

            d1.setEmp(emp);
            emp.setDept(d1);


            em.getTransaction().begin();
            em.persist(d1);
            em.getTransaction().commit();

            System.out.println("done..");
```

--in this case 2 table will be created both will containes the id of each other as FK as an extra column.

department:-(emp_empid as FK)

employee:- (dept_did as FK)

--if we want that only one table should maintains the FK col then we use mappedBy on any side.

ex:-

Department:-

```
    @OneToOne(mappedBy = "dept")
     private Employee emp;
```

--here Employee class maintains the FK id by name dept_did

--if we want to change this FK column name then

```
     @OneToOne
     @JoinColumn(name = "did_FK")
     private Department dept;
```

ex:-

Navigating from dept to emp:-

```
      Department d= em.find(Employee.class, 2).getDept();

           System.out.println(d);
```

Inheritence Mapping:-
=====================

--In DataAccessLayer,between persistent class IS-A relationship is posibly exist.

--but in DB we don't have IS-A relationship between corresponding tables.

--to solve this problem we use inheritence mapping in HB.

JPA supports inheritence mapping with 3 strategy:-

1.one table for entire hirarchy/Single table.

2.table per sub-classes/Joined Table

3.Table per concreate class/ Table Per class:-

1.one table for entire hirarchy/Single table:

--this strategy is the default strategy in HB to perform the inheritance mapping

here we will take a single table with the all the columns, corresponding to generalized properties of super class and specialized properties for all the sub classes and one extra discriminator column.

--with the help of this descriminator value DB table maintains which Entity class of the inheritence hirarcy inserting the record.

Example:

Employee.java:-

```
@Entity
@Inheritence(strategy=InheritanceType.SINGLE_TABLE) // this line is
optional, it is the default strategy
public class Employee {

      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int eid;
      private String name;

//getters and setters


}
```

ContractualEmployee.java:-

```
@Entity
public class ContractualEmployee extends Employee{

      private int noOfWorkingDays;
      private int costPerDay;

//setters and getters
}
```

```
SalaryEmployee.java:-
--------------------

@Entity
public class SalaryEmployee extends Employee{

      private int salary;

//setters and getters

}
```

```
Demo.java:-
--------------

            Employee emp=new Employee();
            emp.setName("Ram");


            SalariedEmployee semp=new SalariedEmployee();
            semp.setName("Mohan");
            semp.setSalary(85000);

            ContractualEmployee cemp=new ContractualEmployee();
            cemp.setName("Hari");
            cemp.setCostPerDay(3000);
            cemp.setNoOfWorkingDays(10);

            em.getTransaction().begin();
            em.persist(emp);
            em.persist(semp);
            em.persist(cemp);
            em.getTransaction().commit();

            System.out.println("done");
```

--here one single table is created with all columns (for all the properties of super class Entity and all the proeperties of all the sub class Entities) plus one extra column DTYPE, which represents which class has made the entry.

--we can change this DTYPE column name and its corresponding value as follows:

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="emptype",discriminatorType=DiscriminatorType.S
TRING)
@DiscriminatorValue(value="emp")
public class Employee {
      int id;
      String name;

//setters and getters

}
```

ContractualEmployee.java:-

```
@Entity
@DiscriminatorValue(value="contEmp")
public class ContractualEmployee extends Employee{


      private int noOfWorkingDays;
      private int costPerDay;

//setters and getters

}
```

SalariedEmployee.java:-

```
@Entity
@DiscriminatorValue(value="salEmp")
public class SalariedEmployee extends Employee{

      private int sal;

//setters and getters

}
```

Note:-the limitation of the above strategy is :-

1.designing a table with huge number of column is not recomended,against the rule of normalizations.

2.with the above strategy,we can not apply not null to the coulmns

2.Table per sub-classes strategy/Joined Table:-
====================================

--in this,every Entity class of inheritence hirarchy will have its own table and these table will participate in relationship,that means every record of child table will represent one record of parent table.

--in this mode of inheritence mapping,each child record of child table maintains association with a record of parent table .

--inside all the child tables we should have a FK column that reffers Pk column of parent table.

--while saving data by using child class obj,the common properies data will be saved to parent table and child class properties will be saved in child table.

Adv of table per subclasses strategy:-

1.DB tables can be designed by satisfying normalization forms/rules.

2.no need to take any discriminator value.

3. not null constraint can be applied.

@Inheritence(strategy=InheritenceType.JOINED) to mention table per child class

@PrimaryKeyJoinColumn(name="PKid") to modify the FK coulmn name in the child class

Example:

Employee.java:-

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Employee {


      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      private int empId;


      private String name;
```

ContractualEmployee.java:-

```
@Entity
@PrimaryKeyJoinColumn(name="eid")
public class ContractualEmployee extends Employee {

      private int noOfWorkingDays;
      private int costPerDay;


}
```

SalariedEmployee.java

```
@Entity
@PrimaryKeyJoinColumn(name="eid")
public class SalariedEmployee extends Employee{

      private int salary;

}
```

3.Table per concreate class/ Table Per class:-
=================================

--in this strategy,every Entity class of inheritence hirarcy will have its own DB table these tables need not stay in relationship.

--in this strategy all the child class corresponding tables has all the column of its super class coresponding columns also.
for ex:-

```
class Employee(id,name)--->employee(id,name);

class SalaryEmployee extends Employee(salary)-------
>semployee(id,name,salary);

class ContractualEmployee extends Employee(noOfWorkingDays,costPerDay)---
------->cemployee(id,name,noOfWorkingDays,costPerDays);
```

--due to this,same column of parent table will be repeated inside all the child table.

Example:

Employee.java:-

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Employee {

      @Id
      @GeneratedValue
      private int empid;

      private String name;


}
```

ContractualEmployee.java:-

```
@Entity
public class ContractualEmployee extends Employee{

      private int noOfWorkingDays;
      private int costPerDay;

}
```

SalariedEmployee.java:-

```
@Entity
public class SalariedEmployee extends Employee{

      private int salary;

}
```

There is another category called MappedSuperclass:-

--Using the MappedSuperclass strategy, we can save only child class object, (here all the data of the child Entity and inherited data of the parent class will be persisted).

--in this strategy parent class will not be an Entity, it will be a normal java class.(can be an abstract class also)

Example

Employee.java:-

```
@MappedSuperclass
public abstract class Employee {


      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      private int empId;


      private String name;
}
```

Demo.java:-

```
            //This emp is not an Entity so it can't be persisted
            //Employee emp=new Employee();
            //emp.setName("Ram");


            SalariedEmployee semp=new SalariedEmployee();
            semp.setName("Mohan");
            semp.setSalary(85000);

            ContractualEmployee cemp=new ContractualEmployee();
            cemp.setName("Hari");
            cemp.setCostPerDay(3000);
            cemp.setNoOfWorkingDays(10);

            em.getTransaction().begin();
            //em.persist(emp);
            em.persist(semp);
            em.persist(cemp);
            em.getTransaction().commit();

            System.out.println("done");
```