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.

Parent class : child class
Parent Entity : Child Entity
Dept class   : Employee class

--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"),inverseJoinColumns=@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.

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.(relationship may not be established)**

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);
```