**EntityManager :**

**find**
**persist**
**remove**
**merge**

**JPQL : Query obj**

**Query q=  em.createQuery(---);**

**int executeUpdate()   --- tx area**
**List getResultList()**
**Object getSingleResult();**

**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+?1 where name=?2";

Query q= em.createQuery(jpql);

q.setParameter(1, 1000);
q.setParameter(2, "Rahul");

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", "Ramesh");

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


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

****Note: For Insert operation we don't use JPQL, we always use persist method of EntityManager.

--whenever we try to project all the columns then the return type of the TypedQuery will be the entire Entity object.

--TypedQuery is used with only one record type of data, if we project List( multiple data) then Query object is enough.

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();

//String jpql ="select name from Account where accno =:ano";
String jpql ="select a.name from Account a where accno =:ano";
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;   // has-A relationship

}
```

//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_PINCODE"))

	})
	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();

	}


}
```