

# Queries (JPQL)

# Java Persistence API (JPA)



# Contents



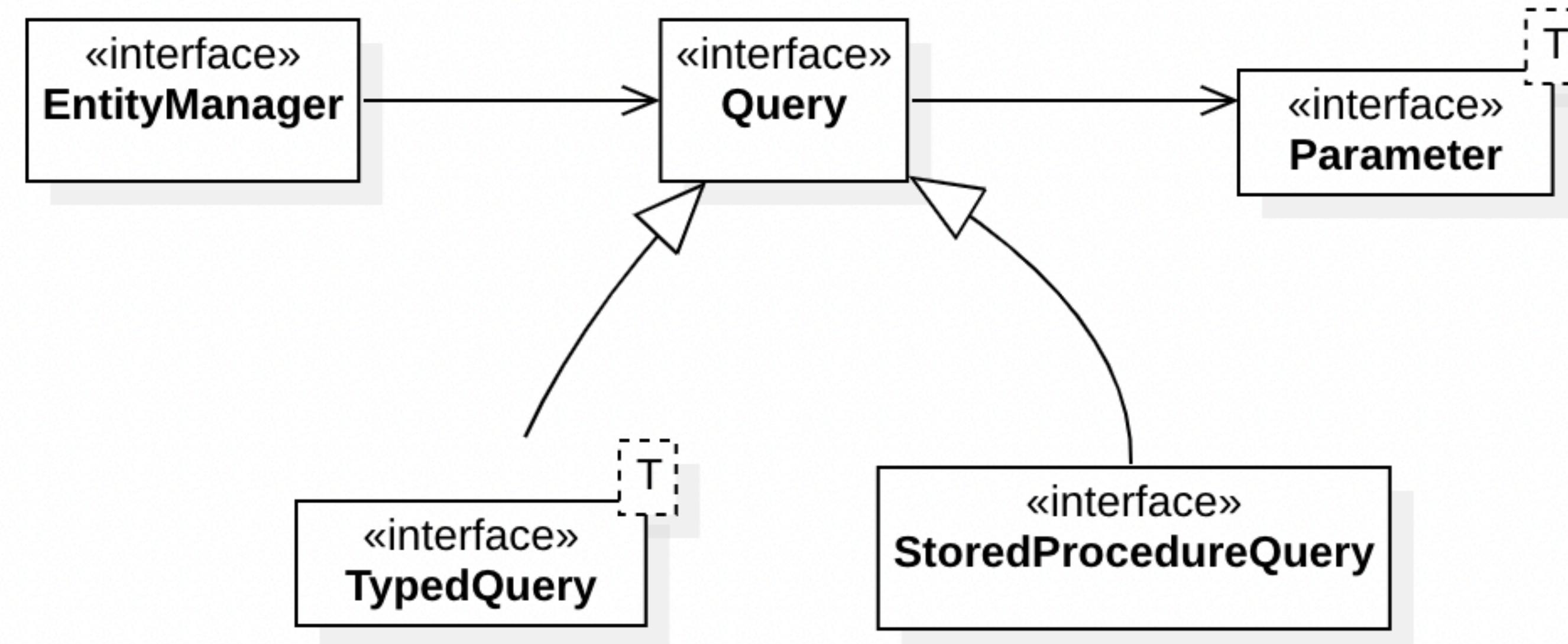
**Hibernate  
JPA**

- About Query
- Types of Query
- Java Persistence Query Language

# Query

- EntityManager ကို အသုံးပြုပြီး Entity တစ်ခုချင်းစီကို Manipulate လုပ်နိုင်ပါတယ်
- ဒါပေမဲ့ Entity တွေကို ရာတဲ့ အခါမှာ ID ကို သိမှသာ ရှာဖွေနိုင်မှာ ဖြစ်သလို့ Manage လုပ်မယ် ဆုံးရင်လဲ Entity တစ်ခုချင်းစီအပေါ်မှာသာ လုပ်ဆောင်နိုင်တာ ဖြစ်ပါတယ်
- လက်တွေ့မှာတော့ ID ကိုသိပြီး ရှာဖွေထားတဲ့ Entity တွေ ကို အသုံးချမှုလဲရှိသလို လိုအပ်တဲ့ Criteria တွေ နဲ့ Entity တွေကို ရှာဖွေတာတို့၊ တစ်ခုထက်မကသော Entity တွေကို Update လုပ်တာတို့ Delete လုပ်တာတို့ လုပ်ဆောင်နိုင်ဖို့လိုအပ်ပါတယ်
- JPA မှာ အထက်ပါ လုပ်ဆောင်ချက်တွေကို ဆောင်ရွက်နိုင်အောင် Query ကို ပြင်ဆင်ထားပါတယ်

# Query Interfaces



EntityManager Interface මා Query තොගී තව්‍යකෙරු ජ්‍යාමත්  
Method තොගී ප්‍රේසෝතිය

# Query & TypedQuery

- Query Statement တွေမှာ အခြေခံအားဖြင့် Database Table ကို Update လုပ်နိုင်တဲ့ Query နှင့် Update မလုပ်ပဲ Data တွေကို Select လုပ်တဲ့ Query ဆိုပီးရိုက်ပါတယ်
- Update လုပ်တဲ့ Statement တွေဆိုရင် Return Value ဟာ Effect ဖြစ်သွားတဲ့ Record Count ဖြစ်တဲ့အတွက် int တန်ဖိုးကို Return ပြန်ပါတယ်
- ဒါပေမဲ့ Select လုပ်တဲ့ Statement ဆိုရင်တော့ Select Clause မှပါတဲ့ တန်ဖိုးအလိုက် Return လုပ်ရမည့် Data Type တွေက မတူက်ပါဘူး
- JPA 2.0 ကနေစပီး Select လုပ်တဲ့ Query တွေမှာ Type Safe ဖြစ်အောင် Generics Type ကို အသုံးပြုတဲ့ TypedQuery ကို ဖြည့်စွက်ခဲ့က်ပါတယ်

# Query Language

Query Language	Description
JPQL	JPA 1.0 ထဲကအသုံးပြုလာခဲ့သော Java Persistence Query Language ဖြစ်ပါတယ်။ SQL နဲ့ ရေးသားပုံဆင်တူသော်လဲ တစ်ချို့နေရာတွေမှာ ကွဲလွှာပါတယ်
Criteria Query	JPA 2.0 ကနေဖြည့်စွက်လာခဲ့တဲ့ Object Query APIs တစ်ခုဖြစ်ပါတယ်။ Dynamic Query များကို ရေးသားတဲ့ နေရာမှာ ကောင်းမွန်ပါတယ်
Native SQL	Native SQL တွေနဲ့ ရေးသားထားတဲ့ Query တွေကိုလဲ JPA မှာ အသုံးပြုနိုင်ပါတယ်
Stored Procedure	Database Programming အမျိုးအစားဖြစ်တဲ့ Stored Procedure တွေကိုအသုံးပြုပြီး JPA Query တွေကို တည်ဆောက်နိုင်ပါတယ်

# Type of Query

Query Language	Static Query	Dynamic Query
JPQL	@NamedQuery	Runtime မှာ JPQL ကို တည်ဆောက် အသုံးပြု
Criteria Query	မရှိပါ	CriteriaQuery Object ကို အသုံးပြုတည်ဆောက်နိုင်
Native SQL	@NamedNativeQuery	Runtime မှာ SQL ကို တည်ဆောက် အသုံးပြု
Stored Procedure	@NamedStoredProcedureQuery	Runtime မှာ StoreProcedureQuery ကို တည်ဆောက် အသုံးပြု

# Using Query



Hibernate  
JPA

- Create Query Object
- Execute Query
- Setting Parameter if required
- Setting Result Ranges if required

# Create JPQL Query

Method	Return Type & Definition
<b>createQuery(String)</b>	Query JPQL နဲ့ရေးသားထားတဲ့ Query String ကနဲ့ Query Object ကို တည်ဆောက်
<b>createQuery(String, Class&lt;T&gt;)</b>	TypedQuery<T> JPQL Query String နဲ့ Result Type ကို အသုံးပြုပြီး TypedQuery ကို တည်ဆောက်
<b>createNamedQuery(String)</b>	Query ကြိုတင်ရေးသားထားတဲ့ NamedQuery ကနဲ့ Query Object ကို တည်ဆောက်
<b>createNamedQuery(String, Class&lt;T&gt;)</b>	TypedQuery<T> NamedQuery နဲ့ Result Type ကို အသုံးပြုပြီး TypedQuery ကို တည်ဆောက်

# Dynamic Query

- Dynamic Query တွေဟာ အရုံးအရှင်းဆုံး Query တွေဖြစ်ပြီး JPQL နဲ့ရေးသားထားသော Query တွေဖြစ်ကြတယ်
- Program အတွင်းလိုအပ်ချက် အလိုက် Runtime မှာ Query တွေကို Build လုပ်နိုင်တယ်
- ဥပမာအားဖြင့် Dynamic Search တွေကို ရေးသားလိုတဲ့ အခါမျိုးမှာ Program နဲ့ Where Clause ကို အမျိုးမျိုးပြောင်းပြီး ရေးသားလေ့ရှိကြပါတယ်
- EntityManager ရဲ့ createQuery method ကို သုံးပြီး Query ကို သော်ငှုံး TypedQuery<T> ကို သော်ငှုံး အသုံးပြုနိုင်ပါတယ်

# Dynamic Query

```
Query query = em.createQuery("SELECT c FROM Customer c");
List<Customer> customers = query.getResultList();
```

```
TypedQuery<Customer> query = em.createQuery("SELECT c FROM Customer c", Customer.class);
List<Customer> customers = query.getResultList();
```

```
String jpqlQuery = "SELECT c FROM Customer c";
if (someCriteria)
    jpqlQuery += " WHERE c.firstName = 'Betty'";
query = em.createQuery(jpqlQuery);
List<Customer> customers = query.getResultList();
```

# Named Query

- တစ်ခါရေးပြီး ပြောင်းလဲစရာမလိုတဲ့ JPQL Statement တွေဆိုရင် Named Query အဖြစ် သတ်မှတ် အသုံးပြန်ပါတယ်
- Named Query တွေကို Entity Class ထဲမှာ @NamedQuery နဲ့ @NamedQueries Annotation တွေကို အသုံးပြုပြီး ရေးသားနိုင်
- @NamedQueries Annotation ထဲမှာ @NamedQuery ကို တစ်ခုထက်မကရေးသားနိုင်ပါတယ်
- @NamedQuery မှာတော့ name နဲ့ query တို့ကို ရေးသားရမှာ ဖြစ်ပါတယ်
- EntityManager ကနေ createNamedQuery Method ကို သုံးပြီး သတ်မှတ်ထားသော Name ကိုပေးကာ Query သို့မဟုတ် TypedQuery Object တို့ကို ရယူနိုင်ပါတယ်

# @NamedQuery

```
12 @Entity
13 @NamedQuery(name="Student.GetAll", query="select s from Student s")
14 public class Student implements Serializable {
15
16     private static final long serialVersionUID = 1L;
17
18     @Id
19     @GeneratedValue(strategy = IDENTITY)
20     private int id;
21
22     private String name;
23     @OneToOne
24     private Contact contact;
25
```

# @NamedQueries

```
13 @Entity
14 @NamedQueries({
15     @NamedQuery(name="Student.GetAll", query="select s from Student s"),
16     @NamedQuery(name="Student.findByName",
17                 query="select s from Student s where s.name = ?1")
18 })
19 public class Student implements Serializable {
20
21     private static final long serialVersionUID = 1L;
22
23⊕     @Id
24     @GeneratedValue(strategy = IDENTITY)
25     private int id;
26
27     private String name;
28⊕     @OneToOne
29     private Contact contact;
```

# Using Named Query

```
@Test  
public void test2() {  
  
    TypedQuery<Student> query = em.createNamedQuery("Student.GetAll", Student.class);  
    List<Student> list = query.getResultList();  
  
    assertNotNull(list);  
  
}
```

# Getting Result

Method	Return Type & Definition
<b>getSingleResult()</b>	T Query ကို Execute လုပ်ပြီးရလာမည့် Result ဟာ Single Object ဖြစ်နေတဲ့ အခါမျိုးမှာ အသုံးပြနိုင်ပါတယ်
<b>getResultList()</b>	List<T> Query ကို Execute လုပ်ပြီးရလာမည့် Result ဟာ Collection ဖြစ်နေတဲ့ အခါမျိုးမှာ အသုံးပြနိုင်ပါတယ်
<b>getResultStream()</b>	Stream<T> Query ကို Execute လုပ်ပြီးရလာမည့် Result ကို Stream အနေနဲ့ ရယူလိုတဲ့ အခါမျိုးမှာ အသုံးပြနိုင်ပါတယ်
<b>executeUpdate()</b>	int Database ကို Effect ဖြစ်စေတဲ့ Query မျိုးတွေကို Execute လုပ်လိုတဲ့ အခါမှာ အသုံးပြနိုင်ပါတယ်

# Query Params

- JPQL ကို ရေးသားရာတွင် ရေးသားပုံ ရေးသားနည်း ၂ မျိုးရှိပါသည်
- Index နံပါတ်ကို အသုံးပြုသောနည်းနှင့် Parameter Name ကို အသုံးပြုသော နည်း ဖြစ်သည်
- ရှေ့နမူနာတွင် ရေးသားခဲ့သော ?1 ဟု ရေးသားခဲ့သည်မှာ Index နံပါတ်ကို အသုံးပြုသောနည်း ဖြစ်ပါသည်
- နံပါတ်ကို 1 ကနေ အစီအစဉ်လိုက်ရေးသားရပါမည်
- Parameter ကို Set လုပ်သော အခါတွင်လဲ Index နံပါတ်အလိုက် Set လုပ်ရမည် ဖြစ်ပါသည်

# Using Parameter Index

# Using Parameter Name

```
13 @Entity  
14 @NamedQueries({  
15     @NamedQuery(name="Student.GetAll", query="select s from Student s"),  
16     @NamedQuery(name="Student.findByName",  
17         query="select s from Student s where s.name = :name")  
18 })  
19 public class Student implements Serializable {  
20 }
```

```
@Test  
public void test2() {  
  
    TypedQuery<Student> query = em.createNamedQuery("Student.findByName", Student.class);  
    query.setParameter("name", "Aung Aung");  
    List<Student> list = query.getResultList();  
  
    assertNotNull(list);  
  
}
```

# JPQL

- Java Persistence Query Language ရဲ့ အတိုကောက် အခေါ်အဝေါ်ဖြစ်ပြီး Entity Object တွေအတွက် Query Language တစ်ခု ဖြစ်ပါတယ်
- Relational Database မှာ သုံးတဲ့ SQL နဲ့ သဘောတရားမှာ အလွန်ဆင်တူပါတယ်
- SQL ဟာ Table တွေ Columns တွေ ကို သုံးပြီး Query ကို ရေးသားပေမဲ့ JPQL ဟာ Entity Object တွေနဲ့ သူတို့ရဲ့ Member တွေ ကိုသုံးပြီး Query ကို ရေးသားရပါမယ်
- JPQL ဟာ Entity Object တွေရဲ့ Member တွေကို Access လုပ်လိုတဲ့အခါမှာ (Dot [.]) ကို သုံးပြီး ဆက်သွယ်ရေးသားလေ့ရှိပါတယ်
- JPQL နဲ့ရေးသားထားတဲ့ Query တွေကို Query Object တွေမှ တဆင့် Execute လုပ်ရမှာ ဖြစ်ပါတယ်
- JPA Framework ကနေ JPQL တွေကို SQL တွေ အဖြစ်ပြောင်းကာ JDBC ကို သုံးပြီး Execute လုပ်မှာ ဖြစ်ပါတယ်

# Select Statement

```
SELECT <select clause>
FROM <from clause>
[WHERE <where clause>]
[ORDER BY <order by clause>]
[GROUP BY <group by clause>]
[HAVING <having clause>]
```

# select clause

```
SELECT [DISTINCT] <expression> [[AS] <identification variable>]  
expression ::= { NEW | TREAT | AVG | MAX | MIN | SUM | COUNT }
```

```
SELECT c.firstName, c.lastName  
FROM Customer c
```

```
SELECT c.address.country.code  
FROM Customer c
```

```
SELECT COUNT(c)  
FROM Customer c
```

# WHERE Clause

```
SELECT c  
FROM Customer c  
WHERE c.age > 18
```

```
SELECT c  
FROM Customer c  
WHERE c.age NOT BETWEEN 40 AND 50
```

```
SELECT c  
FROM Customer c  
WHERE c.address.country IN ('USA', 'Portugal')
```

```
SELECT c  
FROM Customer c  
WHERE c.age = (SELECT MIN(cust. age) FROM Customer cust))
```

# Operators

- `=, >, >=, <, <=, <>`
- [NOT] BETWEEN
- [NOT] LIKE
- [NOT] NULL
- IS [NOT] EMPTY
- [NOT] MEMBER [OF]

# ORDER BY

```
SELECT c
FROM Customer c
WHERE c.age > 18
ORDER BY c.age DESC
```

Multiple expressions may also be used to refine the sort order.

```
SELECT c
FROM Customer c
WHERE c.age > 18
ORDER BY c.age DESC, c.address.country ASC
```

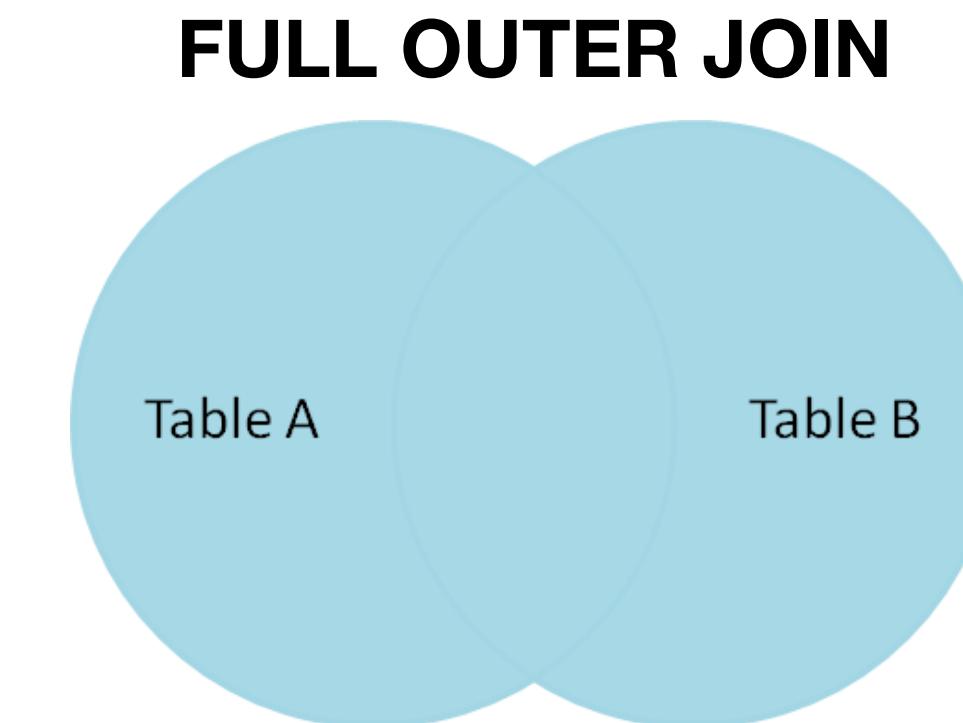
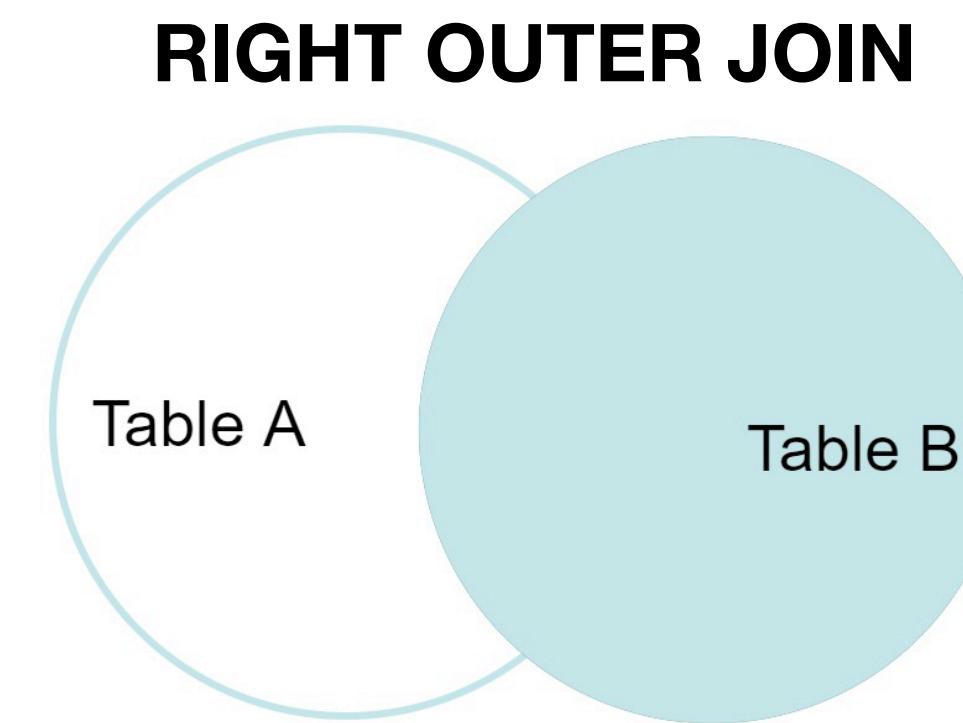
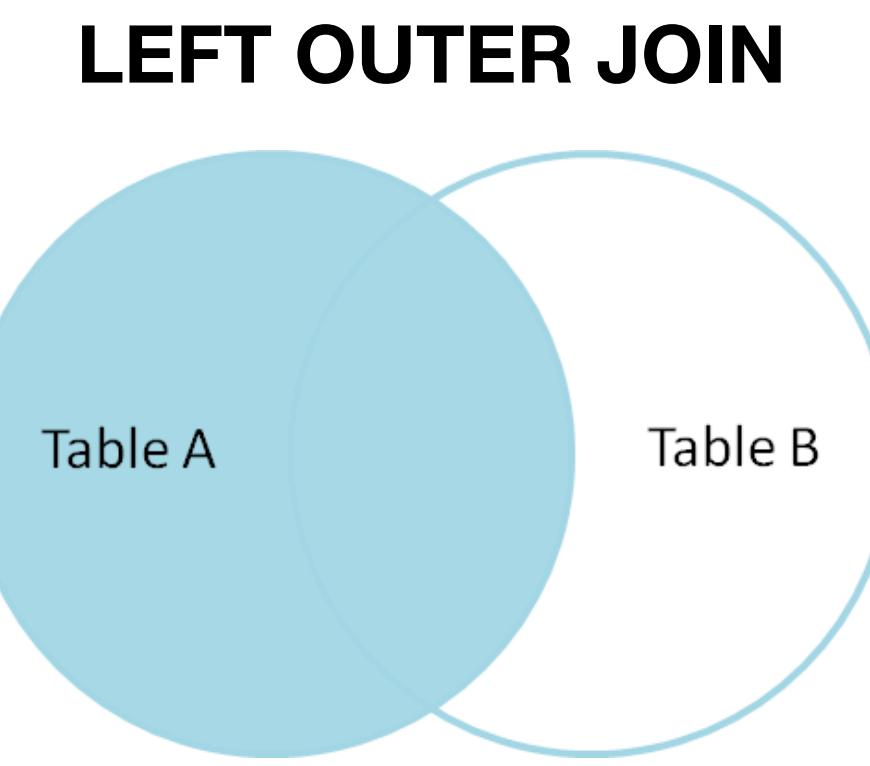
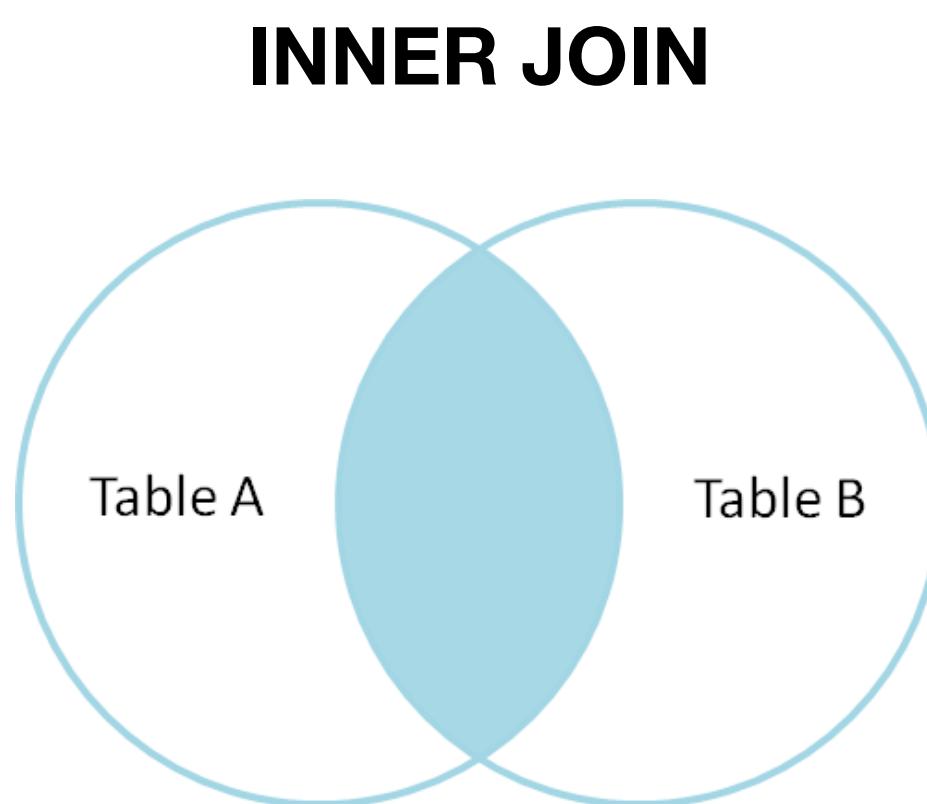
# JOIN in JPA

- JPA Entity တွေမှာ အခြား Entity တွေကို Reference လုပ်ပြီး Relationship တွေကို သတ်မှတ်ရေးသားနိုင်ပါတယ်
- Entity တွေရဲ့ Relationship တွေမှာ Default အတိုင်းကို Table တွေကို Join ပေးသားနိုင်ပါတယ်
- Relationship တွေရဲ့ Fetch Mode, Optional Status အပေါ်မှာမူတည်ပြီး Join လုပ်တဲ့ ပုံစံတွေမှာ ကွဲခြားမှုတွေရှိနိုင်ပါတယ်
- JPQL တွေကို ရေးသားတဲ့ နေရာမှာ JOIN Operator ကို ရေးသားပြီး ဘယ်လို Join လုပ်မလဲ ဆိုတာကို သတ်မှတ်နိုင်ပါတယ်
- JPA ရဲ့ JOIN ကို မလေ့လာခင် SQL တွေမှာ အသုံးပြုတဲ့ JOIN Operator တွေအကြောင်းကို လေ့လာသွားကြရအောင်

# JOIN in SQL

- SQL Select Statement တွေမှာ JOIN Operator ကို အသုံးပြန်ပါတယ်
- JOIN Operator က နာမည်အတိုင်း TABLE နှစ်ခုကို JOIN ပြီး Data တွေကို ဆွဲထုတ်ပေးနိုင်ပြီး JOIN TYPE အလိုက် Fetch လုပ်မည့် data တွေကတော့ မတူကြပါဘူး
- JOIN Operator တွေမှာ အခြေခံအားဖြင့် CROSS JOIN, INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN ဆိုပြီး ရှုကြပါတယ်
- JOIN TYPE အလိုက် ဘယ်လို Data တွေကို Fetch လုပ်မှာလဲ ဆိုတာကို ဆက်ပြီးလွှဲကြည့်ကြရအောင်

# Join Types



# Sample Data

```
● ○ ● Desktop — mysql -u joinusr -p — 91x23
mysql> select * from product;
+----+-----+-----+
| id | name      | category_id |
+----+-----+-----+
| 1  | Potato Chips |      1   |
| 2  | Fish Chips   |      1   |
| 3  | Coke         |      2   |
| 4  | Pepsi        |      2   |
| 5  | iPhone 13    |    NULL  |
[----+-----+-----+]
5 rows in set (0.00 sec)

mysql> select * from category;
+----+-----+
| id | name     |
+----+-----+
| 2  | Drinks   |
| 1  | Foods    |
| 3  | Mobiles  |
+----+-----+
3 rows in set (0.00 sec)

mysql>
```

# CROSS JOIN (cartesian product)

```
Desktop — mysql -u joinusr -p — 91x23
[mysql]> select * from product p join category c;
+----+-----+-----+----+-----+
| id | name           | category_id | id | name       |
+----+-----+-----+----+-----+
| 1  | Potato Chips   |          1 | 3 | Mobiles    |
| 1  | Potato Chips   |          1 | 1 | Foods      |
| 1  | Potato Chips   |          1 | 2 | Drinks     |
| 2  | Fish Chips     |          1 | 3 | Mobiles    |
| 2  | Fish Chips     |          1 | 1 | Foods      |
| 2  | Fish Chips     |          1 | 2 | Drinks     |
| 3  | Coke            |          2 | 3 | Mobiles    |
| 3  | Coke            |          2 | 1 | Foods      |
| 3  | Coke            |          2 | 2 | Drinks     |
| 4  | Pepsi           |          2 | 3 | Mobiles    |
| 4  | Pepsi           |          2 | 1 | Foods      |
| 4  | Pepsi           |          2 | 2 | Drinks     |
| 5  | iPhone 13      |      NULL | 3 | Mobiles    |
| 5  | iPhone 13      |      NULL | 1 | Foods      |
| 5  | iPhone 13      |      NULL | 2 | Drinks     |
+----+-----+-----+----+-----+
15 rows in set (0.00 sec)

mysql>
```

# INNER JOIN

```
Desktop — mysql -u joinusr -p — 91x13

[mysql]> select * from product p join category c on p.category_id = c.id;
+----+-----+-----+----+-----+
| id | name      | category_id | id | name      |
+----+-----+-----+----+-----+
| 3  | Coke       |          2 | 2  | Drinks    |
| 4  | Pepsi      |          2 | 2  | Drinks    |
| 1  | Potato Chips |          1 | 1  | Foods     |
| 2  | Fish Chips  |          1 | 1  | Foods     |
+----+-----+-----+----+-----+
4 rows in set (0.01 sec)

mysql>
```

# LEFT OUTER JOIN

```
Desktop — mysql -u joinusr -p — 91x13

mysql> select * from product p left outer join category c on p.category_id = c.id;
+----+-----+-----+----+-----+
| id | name      | category_id | id   | name    |
+----+-----+-----+----+-----+
| 1  | Potato Chips |           1 |     1 | Foods   |
| 2  | Fish Chips   |           1 |     1 | Foods   |
| 3  | Coke          |           2 |     2 | Drinks  |
| 4  | Pepsi         |           2 |     2 | Drinks  |
| 5  | iPhone 13    |           NULL | NULL | NULL    |
+----+-----+-----+----+-----+
5 rows in set (0.01 sec)

mysql>
```

# RIGHT OUTER JOIN

```
Desktop — mysql -u joinusr -p — 91x13
mysql> select * from product p right outer join category c on p.category_id = c.id;
+----+-----+-----+----+-----+
| id | name      | category_id | id | name    |
+----+-----+-----+----+-----+
| 3  | Coke       |           2 | 2 | Drinks   |
| 4  | Pepsi      |           2 | 2 | Drinks   |
| 1  | Potato Chips |           1 | 1 | Foods    |
| 2  | Fish Chips  |           1 | 1 | Foods    |
| NULL | NULL      |           NULL | 3 | Mobiles  |
+----+-----+-----+----+-----+
5 rows in set (0.00 sec)

mysql>
```

# FULL OUTER JOIN

```
Desktop — mysql -u joinusr -p — 102x19

mysql> select * from product p right outer join category c on p.category_id = c.id
      -> union all select * from product p left outer join category c on p.category_id = c.id;
+-----+-----+-----+-----+
| id   | name    | category_id | id   | name    |
+-----+-----+-----+-----+
| 3    | Coke    |           2 | 2    | Drinks  |
| 4    | Pepsi   |           2 | 2    | Drinks  |
| 1    | Potato Chips | 1 | 1    | Foods   |
| 2    | Fish Chips  | 1 | 1    | Foods   |
| NULL | NULL    | NULL        | 3    | Mobiles |
| 1    | Potato Chips | 1 | 1    | Foods   |
| 2    | Fish Chips  | 1 | 1    | Foods   |
| 3    | Coke    |           2 | 2    | Drinks  |
| 4    | Pepsi   |           2 | 2    | Drinks  |
| 5    | iPhone 13  | NULL        | NULL | NULL    |
+-----+-----+-----+-----+
10 rows in set (0.00 sec)

mysql>
```

MySQL Database မှာ FULL OUTER JOIN ကို Support မလုပ်တဲ့အတွက် LEFT နဲ့ RIGHT JOIN ကို UNION လုပ်  
ပေးရမှာဖြစ်ပါတယ်

# JOIN in Entities Relationships

Relationship	Optional	Fetch	DEFAULT JOIN TYPE
@OneToOne @ManyToOne	TRUE	EAGER	LEFT JOIN
	FALSE	EAGER	INNER JOIN
	TRUE	LAZY	-
	FALSE	LAZY	-
@OneToMany	-	EAGER	LEFT JOIN
	-	LAZY	-
@ManyToMany	-	EAGER	LEFT JOIN
	-	LAZY	-

# JOIN in JPQL

- JPQL Statement တွေကို ရေးသားတဲ့ အခါမှာ Relationship ရှိတဲ့ Entity တွေရဲ့ JOIN ကို  
သတ်မှတ်ရေးသားနိုင်ပါတယ်
- One နဲ့ ဆုံးတဲ့ Relationship တွေမှာကတော့ Join ကို မရေးထားရင်လဲ Default Join Nature  
အလိုက်ရှိနေပြီး၊ Join Type ကို ပြောင်းလဲသတ်မှတ်လိုတဲ့ အခါမျိုးတွေမှာ Join Clause ကို  
ရေးသားနိုင်မှာ ဖြစ်ပါတယ်
- Many နဲ့ ဆုံးတဲ့ Relationship တွေမှာတော့ Reference Entity ထဲက Field ကို Where Clause ထဲ  
မှာ ရေးသားလိုတဲ့ အခါတွေမှာ မဖြစ်မနေ Join ကို ရေးသားပေးရမှာ ဖြစ်ပါတယ်

# To One Relationship



```
var jpql = "select p from Product p where lower(p.category.name) like lower(:name)";

var query = em.createQuery(jpql, Product.class);
query.setParameter("name", "foo".concat("%"));
var result = query.getResultList();

result.forEach(Product::getName);
```

- To One နဲ့ဆုံးတဲ့ Relationship တွေမှာဆိုရင် Default အတိုင်းကို Join ထားပြီးသားဖြစ်ပါတယ်
- Where Clause ထဲမှာ Reference Entity ထဲက Field တွေကို ရေးသားလိုရင်လဲ p.category.name ဆိုပြီး Reference လုပ်ပြီးရေးသားနိုင်ပါတယ်
- ရလဒ်အနေနဲ့ INNER JOIN ကို အသုံးပြုသွားမှာ ဖြစ်ပါတယ်

# Join in To One Relationship



```
var jpql = "select p from Product p join p.category c where lower(c.name) like lower(:category)";

var query = em.createQuery(jpql, Product.class);
query.setParameter("category", "foo%");
var result = query.getResultList();

result.stream().map(Product::getName).forEach(System.out::println);
```

- To One နဲ့ဆုံးတဲ့ Relationship တွေမှာ နှစ်သက်ရာ Join Type ကို သတ်မှတ်ပြီး အသုံးပြုလိုပါက from clause ထဲမှာ join ကို ရေးသားပေးနိုင်ပါတယ်
- Outer Join ကို အသုံးပြုတဲ့ အခါမှာ Data မပါတာတွေရှိနိုင်တဲ့ အတွက် သတိထားပြီး ရေးသားသင့်ပါတယ်

# Join in To Many Relationship

```
var jpql = "select p from Product p join p.supplier s where lower(s.name) like lower(:supplier)";

var query = em.createQuery(jpql, Product.class);
query.setParameter("supplier", "196%");
var result = query.getResultList();

result.stream().map(Product::getName).forEach(System.out::println);
```

- ToMany နဲ့ဆုံးတဲ့ Relationship တွေမှာတော့ Eager Fetch လို့ သတ်မှတ်ထားရင် Default အတိုင်းကို Join ပေးပါတယ
- Where Clause ထဲမှာ Reference Entity ထဲက Field တွေကို ရေးသားလိုက်တော့ Join ပြီးမှ အသုံးပြုရမှာ ဖြစ်ပါတယ

# Projection in JPQL

```
var jpql = "select s.id, s.name, s.phone from Supplier s join s.product p where p.name = :product";
var query = em.createQuery(jpql, Object[].class);
query.setParameter("product", "Potato Chips");

List<Object[]> list = query.getResultList();
```

- Select Clause ကို ရေးသားတဲ့ နေရာမှာ Select လုပ်လိုတဲ့ Entity Field တွေကို စုပြီးရေးသားနိုင်ပါတယ်
- Projection ကို ရေးသားထားတဲ့ Query ရဲ့ Result Type ဟာ Object [] ဖြစ်ပါတယ်
- Projection လုပ်ထားတဲ့ Fields တွေကို အသုံးပြုပြီး DTO အနေနဲ့လက်ခံနိုင်ဖို့အတွက် NEW Operator ကို အသုံးပြုနိုင်ပါတယ်

# NEW Operator

```
var jpql = """
    select new com.jdc.join.demo.dto.SupplierDto(s.id, s.name, s.phone)
    from Supplier s join s.product p where p.name = :product
    """;
var query = em.createQuery(jpql, SupplierDto.class);
query.setParameter("product", "Potato Chips");

var list = query.getResultList();
```

- Select Clause မှာ Projection လုပ်ထားတဲ့ Field တွေအတိုင်း Constructor ပါတဲ့  
Data Class တွေကိုရေးသားထားပါက JPQL Statement ထဲမှာ NEW Operator ကို  
အသုံးပြုပြီး Result Data Type ကို Object [] အတူ DTO အနေနဲ့ ရယူနိုင်ပါတယ်

# Aggregation

- SQL တွေမှာလိုပဲ JQPL ခဲ့ Select Clause တွေမှာ count(), min(), max(), sum(), avg() အစရိတဲ့ Aggregate Function တွေကို ရေးသား အသုံးပြန်ပါတယ်
- Projection လုပ်မည့် Field ဟာ တစ်ခုထဲဆိုပါက Aggregate Function တွေကို ဒီအတိုင်း အသုံးပြန်ပေမဲ့၊ အခြား Field တွေအပေါ်မှာ Group ဖွဲ့ပြီး Aggregate Function တွေကို အသုံးပြုမယ်ဆိုရင် GROUP BY နဲ့ တွဲဖက် အသုံးပြရမှာ ဖြစ်ပါတယ်

# Single Field Aggregation



```
var jpql = "select count(p) from Product p";
var query = em.createQuery(jpql, Long.class);
var count = query.getSingleResult();
```

- Single Field တွေကို Aggregate Function တွေနဲ့ တွဲဖက် အသုံးပြန်ပါတယ်
- Aggregate Function တွေရဲ့ ရလဒ်ဟာ long type ဖြစ်ပါတယ်
- Number Type Field တွေမှာပဲ sum(), avg() တို့ကို အသုံးပြန်ပါတယ်
- Compare လုပ်လို့ရတဲ့ Type တွေမှာပဲ min(), max() တို့ကို အသုံးပြန်မှာ ဖြစ်ပါတယ်

# Grouping and aggregate



```
var jpql = "select p.category.name, count(p) from Product p group by p.category.name";
var query = em.createQuery(jpql, Object[].class);
var list = query.getResultList();
```

- Select ရလဒ်တွေကို Grouping ဖွံ့ဖြိုး Aggregate လုပ်လိုတဲ့ အခါတွေမှာ GROUP BY Clause ကို ရေးသားနိုင်ပါတယ်
- GROUP BY Clause ထဲမှာ Grouping ဖွံ့လိုတဲ့ Fields တွေကို Comma ခံပြီး ရေးသားပေးရမှာ ဖြစ်ပါတယ်

# HAVING Clause



```
var jpql = """
    select p.category.name, count(p) from Product p
    group by p.category.name having count(p) < 10
""";
var query = em.createQuery(jpql, Object[].class);
var list = query.getResultList();
```

- Grouping လုပ်ထားတဲ့ တန်ဖိုးတွေကို Filtering လုပ်ချင်တဲ့ အခါမှာ Having Clause မှာ ရေးသားနိုင်ပါတယ်
- Entity Field တွေနဲ့ Filter လုပ်မယ်ဆိုရင်တော့ Where Clause မှာလဲ ရေးသားနိုင်ပေမဲ့ Grouping လုပ်ပြီး Aggregate လုပ်ထားတဲ့ တန်ဖိုးတွေနဲ့ Filter လုပ်ချင်ပြီဆိုရင်တော့ HAVING Clause မှာပဲ ရေးသားနိုင်မှာဖြစ်ပါတယ်

# Bulk Delete

- EntityManager ရဲ့ remove method ကို အသုံးပြုမယ်ဆိုရင် Entity Object တစ်ခုချင်းစီကို Delete လုပ်နိုင်မှာ ဖြစ်သော်လည်း၊ Criteria အပေါ်မူတည်ပြီး အစုလိုက် Delete လုပ်ချင်ရင် JPQL ရဲ့ Delete ကို အသုံးပြုနိုင်မှာ ဖြစ်ပါတယ်

```
DELETE FROM <entity name> [[AS] <identification variable>]  
[WHERE <where clause>]
```

```
DELETE FROM Customer c  
WHERE c.age < 18
```

# Bulk Update

- EntityManager ရဲ့ merge method ကို သုံးခြင်းအားဖြင့်၍၏ Managed State မှာရှိတဲ့ Entity ရဲ့ Fields တွေရဲ့ တန်ဖိုးကို ပြောင်းခြင်းအားဖြင့် သော်၍၏ Entity တစ်ခုရဲ့ တန်ဖိုးတွေကို Update လုပ်နိုင်ပါတယ်
- ဒါပေမဲ့ Entity များကို အစုလိုက် Update လုပ်ချင်ရင်တော့ JPQL ရဲ့ Update Statement ကို အသုံးပြုရမှာ ဖြစ်ပါတယ်

```
UPDATE <entity name> [[AS] <identification variable>]
SET <update statement> {, <update statement>}*
[WHERE <where clause>]
```

```
UPDATE Customer c
SET c.firstName = 'TOO YOUNG'
WHERE c.age < 18
```

# Range of Query Result



```
@NamedQuery(name = "Sales.searchDetails", query = """
    select new com.jdc.query.dto.SaleDetails(
        s.id, s.saleDate, c.name, sum(ps.product.price * ps.quantity))
    from Sale s join s.products ps join s.customer c
    group by s.id, s.saleDate, c.name
""")
public class SalesService {

    private EntityManager em;

    public SalesService(EntityManager em) {
        this.em = em;
    }

    public List<SaleDetails> search(int start, int limit) {
        var query = em.createNamedQuery("Sales.searchDetails", SaleDetails.class);
        query.setFirstResult(start);
        query.setMaxResults(limit);

        return query.getResultList();
    }
}
```