1_JPA-Basic(ORMStandard)

▼ 01. JPA introduction

▼ What is JPA?

Java Persistence API

ORM technical standard of Java

Generally use hibernates as the structure

▼ Difference between Object and RDB

1. Inheritance
2. Relationship
3. Data type
4. The way recognize data

▼ Why should I use JPA?

- OOP
  1. Using object and method rather than writing all the queries
  2. Data are managed as Entity (Object)

- Productivity

  Much shorter code for simple CRUD

  Ex)

  Create: em.persist(member)

  Read: Member member = em.find(memberId)

  Update: member.setName(”newName”)

  Delete: em.remove(member)

- Maintenance

  If new column added, adding equivalent field is only thing to do.

  NO NEED TO MODIFY ALL THE QUERIES.

- Resolving paradigm mismatch

  Object (in Java) vs Supper and sub type (in RDB)

  The mismatch can be resolved by relationship mapping

- Performance
  1. Ensuring identity(sameness) with level 1 cache.

     If exists, it uses the entity in the level 1 cache rather than fetching the data from the DB again.

```
String memberId = "100";
Member m1 = jpa.find(Member.class, memberId); //SQL
Member m2 = jpa.find(Member.class, memberId); //캐시

println(m1 == m2) //true
```

SQL 1번만 실행

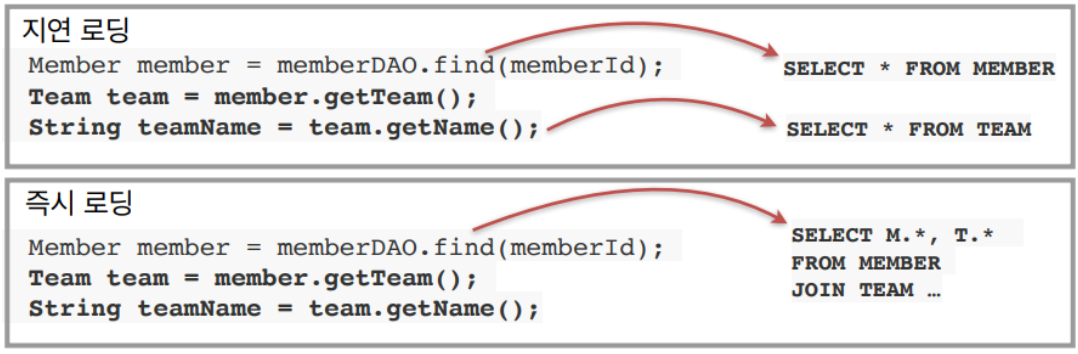
```
Member a = em.find(Member.class, "member1");
Member b = em.find(Member.class, "member1");

System.out.println(a == b); //동일성 비교 true
```

  2. Transactional write-behind

     Many queries are dispatched at once.

  3. Lazy loading

     Relational entity are fetched when needed.



- Data access abstraction and vendor independence
- Standard

▼ 02. JPA start

Version: Java 8

Build tool: Maven

▼ DB Dialect

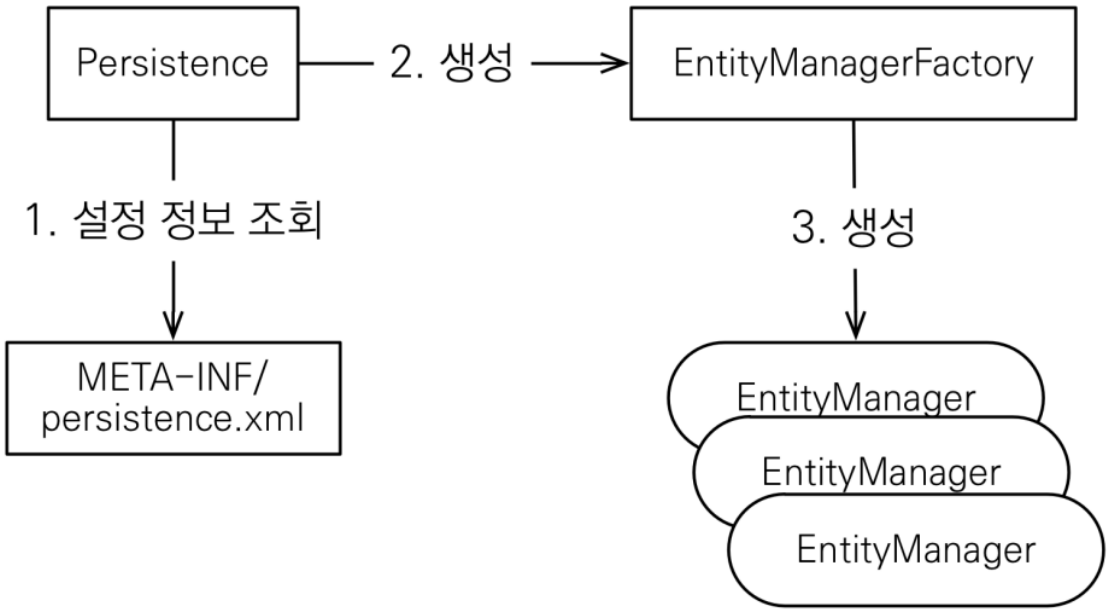
JPA is not dependent on specific database (ROWNUM, LIMIT et cetera)

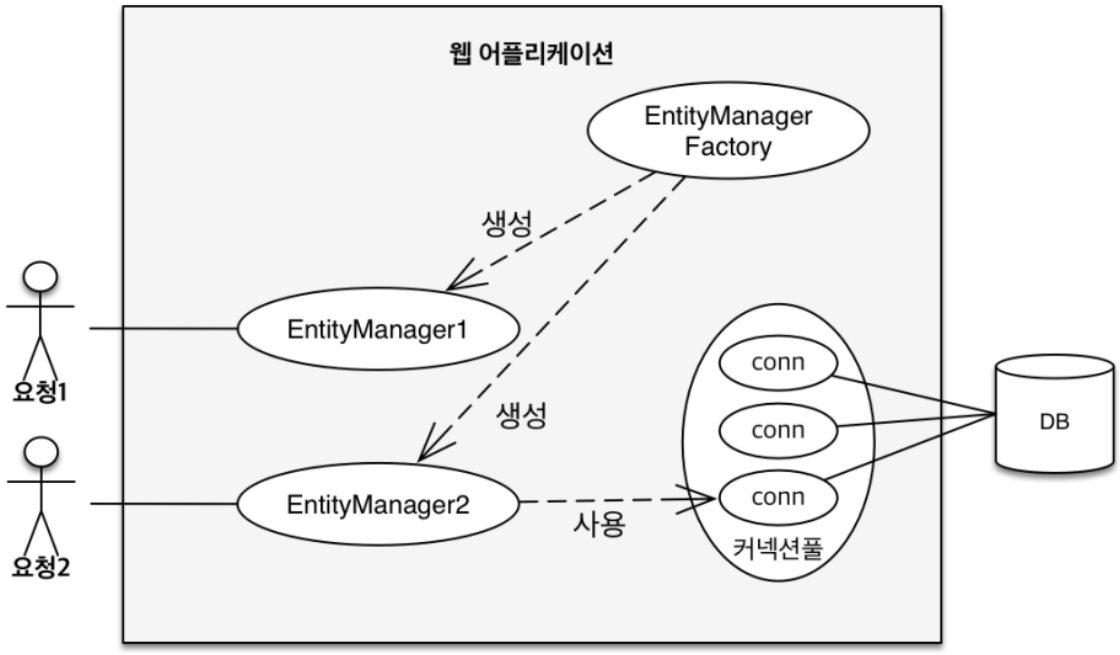
- dialect setting

```
<property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
```

▼ How JPA runs

1. One application, one EntityManagerFactory
2. One tread, one EntityManager
3. All data modifications are executed in the transaction.





03. Persistence management

▼ 03. Persistence management

▼ Persistence context (엔티티 지속 환경)

Logical concept

Deal with it by EntityManger

- context

  The circumstances that form for an event, statement, or idea.

▼ Entity life cycle

- Non-persistence

  new

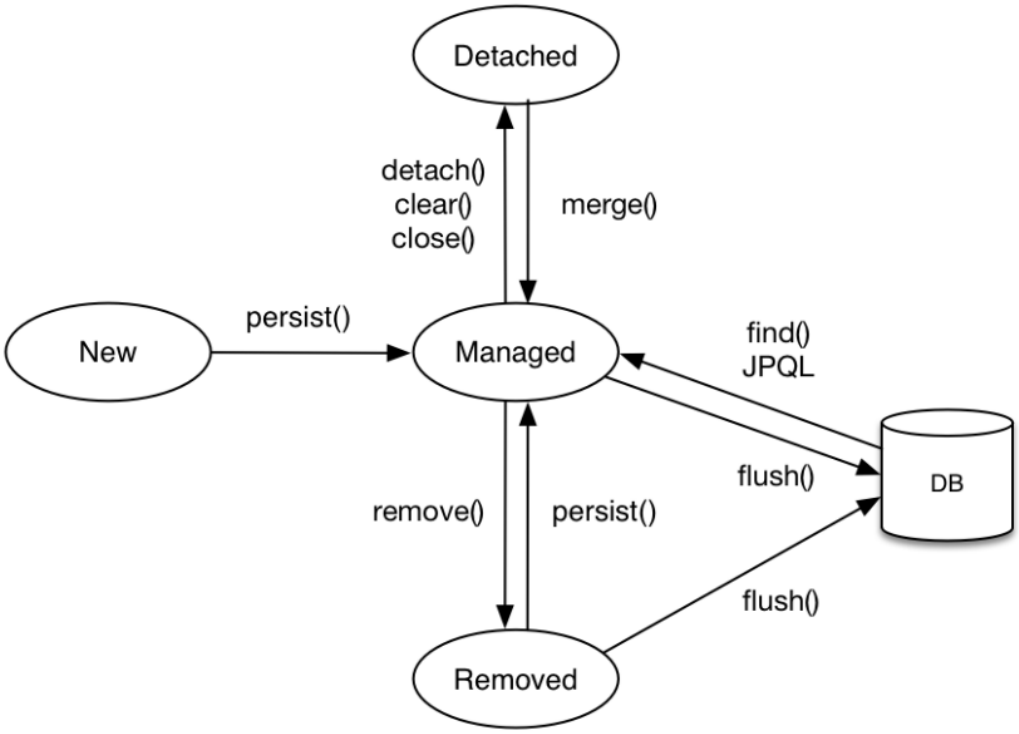
- Persistence

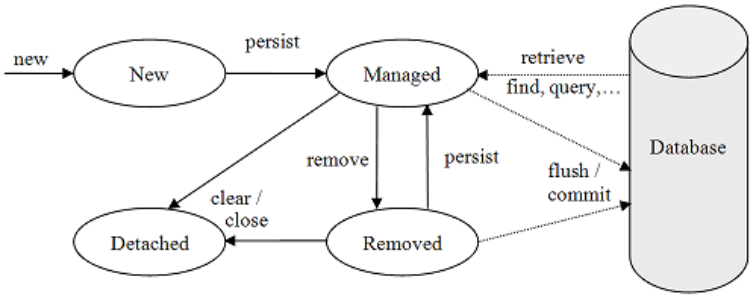
  em.persist(obj);

- quasi-persistence

the state an entity is detached from persistence context

- em.detatch(obj);
- em.clear()
- em.close()





▼ Advantages of persistence context

- Level 1 cache

  If exists, it uses the entity in the level 1 cache rather than fetching the data from the DB again.

```
String memberId = "100";
Member m1 = jpa.find(Member.class, memberId); //SQL
Member m2 = jpa.find(Member.class, memberId); //캐시

println(m1 == m2) //true
```

SQL 1번만 실행

- ensuring identity(sameness)

```
Member a = em.find(Member.class, "member1");
Member b = em.find(Member.class, "member1");

System.out.println(a == b); //동일성 비교 true
```

- Transactional write-behind

  Many queries are dispatched at once

- Dirty checking

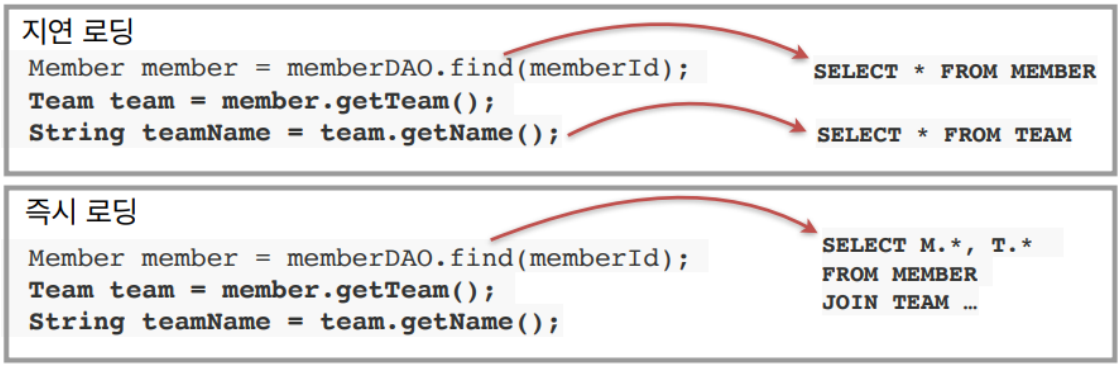
  When and entity is modified, equivalent query is dispatched automatically.

  member.setXxx(“”);

```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
transaction.begin();  // [트랜잭션] 시작

// 영속 엔티티 조회
Member memberA = em.find(Member.class, "memberA");

// 영속 엔티티 데이터 수정
memberA.setUsername("hi");
memberA.setAge(10);

//em.update(member) 이런 코드가 있어야 하지 않을까?

transaction.commit(); // [트랜잭션] 커밋
```

- Lazy loading

  Relational entity are fetched when needed.



▼ Flush

Apply the changes in persistence context to the DB

Synchronize the DB with the persistence context.

▼ Flush ≠ clear

clear: clear the persistence context

▼ Three cases that persistence context are flushed

1. DIrect call

   em.flush()

2. Automatic call

   1) transaction call

      em.commit()

   2) JPQL query call

      em.createQuery()

      flush is called before jpql call because data might not be found due to ‘transactional write-behind’.

```
em.persist(memberA);
em.persist(memberB);
em.persist(memberC);

//중간에 JPQL 실행
query = em.createQuery("select m from Member m", Member.class);
List<Member> members= query.getResultList();
```

▼ flush mode option

em.setFlushMode(FlushModeType.COMMIT)

- FlushModeType.COMMIT

  on commit call or query call

  - em.commit()
  - em.createQuery()

- FlushModeType.COMMIT

  on commit call

▼

▼ 04. Entity mapping

▼ @Entity

the class is managed by JPA

necessary when using jpa

▼ Note

1. Default constructor is necessary (pulic or protected)
2. Not applicable for final class, enum, interface or inner class
3. ‘final’ keyword not applicable for the entity fields

▼ Attribute

- name

Entity name that JPA uses

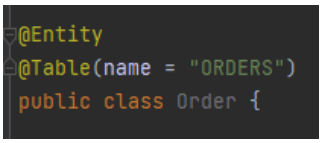
Default: class name

DO NOT CHNAGE THE NAME USING THIS ATTRIBUTE.

IF SAME CLASS NAME EXEIST, CHANGE THE CLASS NAME.

▼ @Table

set mapping table



▼ Attribute

속성	기능	기본값
name	매핑할 테이블 이름	엔티티 이름을 사용
catalog	데이터베이스 catalog 매핑	
schema	데이터베이스 schema 매핑	
uniqueConstraints (DDL)	DDL 생성 시에 유니크 제약 조건 생성	

▼ DDL: Database schema auto create

▼ persistence.xml (ddlauto)

```
<property name="hibernate.hbm2ddl.auto" value="create" />
```

hibernate.hbm2ddl.auto

옵션	설명
create	기존테이블 삭제 후 다시 생성 (DROP + CREATE)
create-drop	create와 같으나 종료시점에 테이블 DROP
update	변경분만 반영(운영DB에는 사용하면 안됨)
validate	엔티티와 테이블이 정상 매핑되었는지만 확인
none	사용하지 않음

- Staging or Production server

  validate or none

*** NEVER USE create, create-drop, update on production equipment

- Test server

  update or validate

- the early stage of development

  create or update

▼ DDL CREATE FUNTION

It does not affect JPA logic, but only used when creating DDL automatically

Nevertheless, it good to state clearly like following for fellow programmers to easily notify the structure.

▼ @Column

@Column(nullable - fasle, length = 10)

▼ unique constraint

```
@Table(uniqueCOnstraints = {@UniqueConstraint(name = “NAME_AGE_UNIQUE”,
                                              columnNumes={"NAME", "AGE"})})
```

▼ Fields and columns

```
@Entity
public class Member {
 @Id
 private Long id;

 @Column(name = "name")
 private String username;
 private Integer age;

 @Enumerated(EnumType.STRING)
 private RoleType roleType;

 @Temporal(TemporalType.TIMESTAMP)
 private Date createdDate;

 @Temporal(TemporalType.TIMESTAMP)
 private Date lastModifiedDate;

 @Lob
 private String description;

}
```

Refer to the pdf

▼ 05. relationship mapping - basic

- 단방향
- 양방향
- 양방향 주의점

  https://raejin.tistory.com/209

▼ 06. various relationship mapping

- N:1
- 1:N
- 1:1
- N:N

▼ 07. Advanced mapping

▼ Table strategies

@Inheritance(strategy=InheritanceType.XXX)

- **JOINED (Recommended)**

  Table normalization

- SINGLE_TABLE

  Sometimes in need

  Fast reading time

- TABLE_PER_CLASS

  DO NOT USE

  Both programmers and DBA don’t prefer.

▼ @MappedSuperclass

It is not an Entity

Make it as ‘abstract class’

```
@MappedSuperclass
public abstract class BaseEntity {
```

```
  @Column(name = "INSERT_MEMBER")
  private String createdBy;

  private LocalDateTime createdDate;

  @Column(name = "UPDATE_MEMBER")
  private String lastModifiedBy;
  private LocalDateTime lastModifiedDate;
```

```
@Entity
public class Delivery extends BaseEntity { ... }
```

- @DiscriminatorColumn(name=”XXX”)
- @DiscriminatorValue(”xxx”)

▼ 08. Proxy and relationship management

Use LAZY loading!!

DO NOT USE EAGAR!

▼ 09. Value type

!!! Use as invariable[unchangeable]

1. Java type
   - primitive (int, double)

     DO NOT share primitive type

   - rapper class (Integer, Long)

     invariable[unchangeable] object

   - String
2. Embedded type

```
@Embeddable
public class Period {

  private LocalDateTime startDate;
  private LocalDateTime endDate;
```

```
@Embedded
  private Period period;
```

3. Collection value type

▼ 10. Object oriented query language.

▼ JPQL

▼ Concept

Object oriented query language that abstract SQL that JPA provides

JPQL targets Entity not Table

JPQL doesn’t rely on a specific Database

Similar grammar with SQL

SELECT, FROM, WHERE, GRUOP BY, HAVING, JOIN

Ex)

```
//검색
String jpql = "select m from Member m where m.age > 18";

List<Member> result = em.createQuery(jpql, Member.class)
 .getResultList();
```

```
실행된 SQL
 select
     m.id as id,
     m.age as age,
     m.USERNAME as USERNAME,
     m.TEAM_ID as TEAM_ID
 from
     Member m
 where
     m.age>18
```

▼ TypeQuery, Query

- TypeQuery (specific type)

```
TypedQuery<Member> query =
 em.createQuery("SELECT m FROM Member m", Member.class);
```

- Query (unspecific type)

```
Query query =
 em.createQuery("SELECT m.username, m.age from Member m");
```

▼ Get Result

query.getResultLIst()

query.getSingleResult()

▼ Parameter binding

- Name base

```
SELECT m FROM Member m where m.username=:username
query.setParameter("username", usernameParam);
```

- Location base

DO NOT USE THIS PATTERN. It’s awful when modifying queries

```
SELECT m FROM Member m where m.username=?1
query.setParameter(1, usernameParam)
```

▼ Projection

1. Entity

   SELECT m FROM Member m

   SELECT m.team FROM Member m

2. Embedded type

   SELECT m.address FROM Member m

3. Scala type

   SELECT m.username, m.age FORM Member m

   1) Query

   2) Object[]

   3) new using DTO

      Full path including pakage name

      Equivalent constructor needed

```
SELECT new jpabook.jpql.UserDTO(m.username, m.age)
FROM Member m
```

▼ Paging API

JPA creates paging query based on the dialect

setFirstResult(int startPosition)

setMaxResult(int maxResult)

```
//페이징 쿼리
 String jpql = "select m from Member m order by m.name desc";
 List<Member> resultList =
      em.createQuery(jpql, Member.class)
         .setFirstResult(10)
         .setMaxResults(20)
         .getResultList();
```

Ex)

```
// MySQl
SELECT
 M.ID AS ID,
 M.AGE AS AGE,
 M.TEAM_ID AS TEAM_ID,
 M.NAME AS NAME
FROM
 MEMBER M
ORDER BY
 M.NAME DESC LIMIT ?, ?
```

```
// Oracle
SELECT * FROM
 ( SELECT ROW_.*, ROWNUM ROWNUM_
 FROM
 ( SELECT
 M.ID AS ID,
 M.AGE AS AGE,
 M.TEAM_ID AS TEAM_ID,
 M.NAME AS NAME
 FROM MEMBER M
 ORDER BY M.NAME
 ) ROW_
 WHERE ROWNUM <= ?
 )
WHERE ROWNUM_ > ?
```

▼ Join

▼ Inner join

```
SELECT m FROM Member m [INNER] JOIN m.team t
```

▼ Outer join

```
SELECT m FROM Member m LEFT [OUTER] JOIN m.team t
```

▼ Theta join

```
SELECT count(m) FROM Member m, Team t WHERE m.username
= t.name
```

▼ ON clause

1) Filtering

```
// JPQL
SELECT m, t FROM Member m LEFT JOIN m.team t on t.name = 'A'
```

```
// SQL
SELECT m.*, t.*
FROM Member m LEFT JOIN Team t
  ON m.TEAM_ID=t.id
  and t.name='A'
```

2) Outer join with an entity that there’s no relationship

```
// JPQL
SELECT m, t
FROM Member m LEFT JOIN Team t
  on m.username = t.name
```

```
// SQL
SELECT m.*, t.*
FROM Member m LEFT JOIN Team t
  ON m.username = t.name
```

▼ Explicit | Implied join

USE EXPLICIT JOIN !!

1) Easier to recognize

2) Join is an important point for SQL tunning

3) Implied join affects path searching

- Explicit

```
select m from Member m join m.team t
```

- Implied

INNER JOIN !!

```
select m.team from Member m
```

▼ Subquery

JPA allows subqueries only in 'WHERE and HAVING' clauses.

HIBERNATES allows subqueries in 'SELECT' clause as well.

Subqueries are not possible in 'FROM' clause

Suggested solution

1) use 'JOIN' if possible

2) Get two (or more) results and filter then in java

** In fact, subqueries in FROM clause deteriorate searching performance

▼ Where clause examples

- Member whose age is over the average

```
select m from Member m
where m.age > (select avg(m2.age) from Member m2)
```

- Member who make an order once or more

```
select m from Member m
where (select count(o) from Order o where m = o.member) > 0
```

▼ Comparative grammar

▼ [NOT] EXISTS

팀A 소속인 회원

```
select m from Member m
where exists (select t from m.team t where t.name = '팀A')
```

▼ ALL, ANY, SOME

```
// 전체 상품 각각의 재고보다 주문량이 많은 주문들
select o from Order o
where o.orderAmount > ALL (select p.stockAmount from Product p)
```

```
//어떤 팀이든 팀에 소속된 회원
select m from Member m
where m.team = ANY (select t from Team t)
```

- [NOT] IN
- AND OR NOT
- =, >, >=, <, <=, <>
- BETWEEN, LIKE, IS NULL

▼ JPQL data types

문자: ‘HELLO’, ‘She’’s’

숫자: 10L(Long), 10D(Double), 10F(Float)

Boolean: TRUE, FALSE

ENUM: jpabook.MemberType.Admin (패키지명 포함)

엔티티 타입: TYPE(m) = Member (상속 관계에서 사용)

▼ CASE clause

- Basic

```
select
 case when m.age <= 10 then '학생요금'
 when m.age >= 60 then '경로요금'
 else '일반요금'
 end
from Member m
```

- Simple

```
select
 case t.name
 when '팀A' then '인센티브110%'
 when '팀B' then '인센티브120%'
 else '인센티브105%'
 end
from Team t
```

▼ Default functions

- CONCAT
- SUBSTRING
- TRIM
- LOWER, UPPER
- LENGTH
- LOCATE
- ABS, SQRT, MOD
- SIZE, INDEX(JPA 용도)

- COALESCE

  사용자 이름이 없으면 이름 없는 회원을 반환

```
select coalesce(m.username,'이름 없는 회원') from Member m
```

- NULLIF

  사용자 이름이 '관리자'면 null을 반환하고 나머지는 본인의 이름을 반환

```
select NULLIF(m.username, '관리자') from Member m
```

▼ User-defined functions

- Declare

  Make class and extends DB

```
package dialect;

import org.hibernate.dialect.H2Dialect;
import org.hibernate.dialect.function.StandardSQLFunction;
import org.hibernate.type.StandardBasicTypes;

public class MyH2Dialect extends H2Dialect {

  public MyH2Dialect() {
    registerFunction("group_concat", new StandardSQLFunction("group_concat", StandardBasicTypes.STRING));
  }
}
```

- Usage

```
select function('group_concat', i.name) from Item i
```

```
select group_concat(i.name) from Item i
```

▼ Path expression

- State field
- Association field

  ▼ Single value

  @ManyToOne, @OneToOne, 대상이 엔티티(ex: m.team)

  Further searching is available

```
// JPQL
select o.member from Order o
```

```
// SQL
select m.*
 from Orders o
 inner join Member m on o.member_id = m.id
```

▼ Collection value

@OneToMany, @ManyToMany, 대상이 컬렉션(ex: m.orders)

Collection is the end of searching. Further searching is NOT available.

→ Possible if it gets a alias in FROM clause.

▼ Examples

select o.member.team from Order o -> 성공

select t.members from Team -> 성공

select t.members.username from Team t -> 실패

select m.username from Team t join t.members m -> 성공

▼ Fetch join

is the function for performance optimization offered by JPQL

Related entities or collections can be selected at once.

'FetchType.LAZY' doesn't work when fetch join is used. → EAGAR roading

▼ Usage tip

1. Set all the global roading strategy 'LAZY'

   → @OneToMany(fetch = FetchType.LAZY).
2. Use fetch join when needed.

- When you shouldn't use fetch join

  If you need to make an different results from entity,

  using general join rather than fetch join is more efficient.

  Then, return the data in DTO.

▼ Limit(ation)

1. Alias can't be given to fetch join target

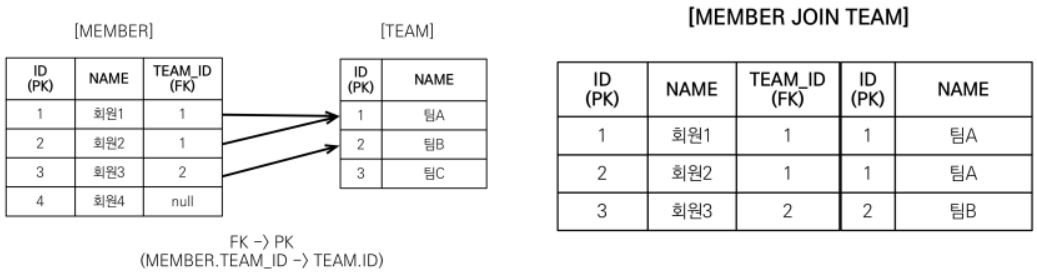
2. More than two collections can’t be fetch joined
3. Paging API(setFirstResult, setMaxResults) can’t be used when jetch join.

   ** Fetch join paging is possible in case of Single value relational fields such as 1:1, N:1

▼ N:1

```
// [JPQL]
select m from Member m join fetch m.team
```

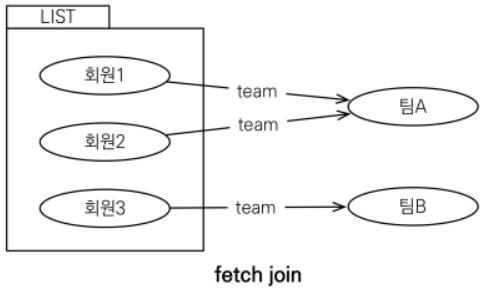
```
// [SQL]
SELECT M.*, T.* FROM MEMBER M
INNER JOIN TEAM T ON M.TEAM_ID=T.ID
```



ID (PK)	NAME	TEAM_ID (FK)
1	회원1	1
2	회원2	1
3	회원3	2
4	회원4	null

ID (PK)	NAME
1	팀A
2	팀B
3	팀C

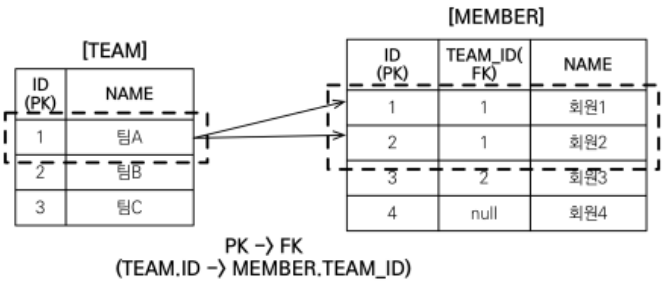
ID (PK)	NAME	TEAM_ID (FK)	ID (PK)	NAME
1	회원1	1	1	팀A
2	회원2	1	1	팀A
3	회원3	2	2	팀B



▼ 1:N

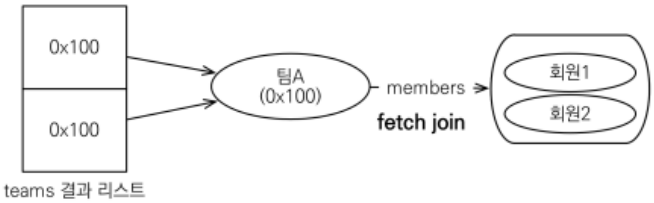
```
[JPQL]
select m from Member m join fetch m.team
```

```
[SQL]
SELECT M.*, T.* FROM MEMBER M
INNER JOIN TEAM T ON M.TEAM_ID=T.ID
```



[TEAM JOIN MEBER]

ID (PK)	NAME	ID (PK)	TEAM_ID (FK)	NAME
1	팀A	1	1	회원1
1	팀A	2	1	회원2



▼ JPQL DISTINCT

1. Deduplicating like SQL
2. Entity deduplication from the application.

▼ Polymorphic query

- Type

```
[JPQL]
select i from Item i
where type(i) IN (Book, Movie
```

```
[SQL]
select i from i
where i.DTYPE in (‘B’, ‘M’)
```

- Treat

```
[JPQL]
select i from Item i
where treat(i as Book).auther = ‘kim’
```

```
[SQL]
select i.* from Item i
where i.DTYPE = ‘B’ and i.auther = ‘kim’
```

▼ Direct use of Entities

If an entity is used as a parameter, JPQL use it’s PK.

- Primary key

```
엔티티를 파라미터로 전
String jpql = “select m from Member m where m = :member”;
List resultList = em.createQuery(jpql)
 .setParameter("member", member)
 .getResultList();
```

```
식별자를 직접 전달
String jpql = “select m from Member m where m.id = :memberId”;
List resultList = em.createQuery(jpql)
 .setParameter("memberId", memberId)
 .getResultList();
```

```
실행된 SQL
select m.* from Member m where m.id=?
```

- Foreign key

```
엔티티를 파라미터로 전
Team team = em.find(Team.class, 1L);
String qlString = “select m from Member m where m.team = :team”;
List resultList = em.createQuery(qlString)
 .setParameter("team", team)
 .getResultList();
```

```
식별자를 직접 전달
String qlString = “select m from Member m where m.team.id = :teamId”;
List resultList = em.createQuery(qlString)
 .setParameter("teamId", teamId)
 .getResultList();
```

```
실행된 SQL
select m.* from Member m where m.team_id=?
```

▼ Named query

It great because query error can be found on compile.

DO NOT use both below.

It can be declared in repository using SpringBoot

1. Declare in an entity using annotation

```
@Entity
@NamedQuery(
 name = "Member.findByUsername",
 query="select m from Member m where m.username = :username")
public class Member {
 ...
}

List<Member> resultList =
 em.createNamedQuery("Member.findByUsername", Member.class)
 .setParameter("username",
"회원1")
 .getResultList();
```

2. XML definition

▼ Bulk Operation

DO NOT USE dirty checking. It requires to many query execution.

** NOTE

Bulk operation ignore persistent context and dispatch query to the database.

So, EntityFactory should be cleared after bulk operation. → em.clear()

```
String qlString = "update Product p " +
                  "set p.price = p.price * 1.1 " +
                  "where p.stockAmount < :stockAmount";

int resultCount = em.createQuery(qlString)
                    .setParameter("stockAmount", 10)
                    .executeUpdate();
```

▼ JPA Criteria

allows to wright JPQL through java codes

works as JPQL builder

Official function from JPA

TOO COMPLICATE AND NOT PRACTICAL

→ using QueryDSL is recommended

▼ QueryDSL

allows to wright JPQL through java codes

works as JPQL builder

error can’t be found on compile

good for dynamic query

easy and simple to use

```
//JPQL
//select m from Member m where m.age > 18

JPAFactoryQuery query = new JPAQueryFactory(em);
QMember m = QMember.member;

List<Member> list =
   query.selectFrom(m)
         .where(m.age.gt(18))
         .orderBy(m.name.desc())
         .fetch();
```

▼ Native SQL

direct use of SQL that JPA offers

offers functions that rely on a specific database which can’t be solved by JPQL (Oracle’s connect by, particular SQL hint etc..)

```
String sql =
 “SELECT ID, AGE, TEAM_ID, NAME FROM MEMBER WHERE NAME = ‘kim’";

List<Member> resultList =
 em.createNativeQuery(sql, Member.class).getResultList();
```

▼ JDBC API (MyBatis, SpringJdbcTemplate)

JPA makes it compatible to use one technology with others such as ‘jdbc connection, spring JdbcTemplate or Mybatis‘.

!!! Persistence context must be flushed on a appropriate time