

1장 JPA 소개

기간 : 21.12.01 ~ 21.12.08

ORM(Object-Relational Mapping) 프레임워크

JPA 장점

1.1 SQL을 직접 다룰 때 발생하는 문제점

1.1.1 작업의 반복

1.1.2 SQL에 의존적인 개발

1.1.3 JPA와 문제 해결

1.2 패러다임의 불일치 40p

패러다임의 불일치로 인해 발생하는 문제

1.2.1 상속 41p

1.2.2 연관관계 43p

1.2.3 객체 그래프 탐색 48p

1.2.4 비교 51p

1.3 JPA(Java Persistence API)란? 54p

ORM이란?

1.3.1 JPA 소개 56p

1.3.2 왜 JPA를 사용해야 하는가? 57p

ORM(Object-Relational Mapping) 프레임워크

- 객체와 관계형 데이터베이스 간의 차이를 중간에서 해결해주는 프레임워크

JPA 장점

- 애플리케이션을 SQL이 아닌 객체 중심으로 개발하니 생산성과 유지보수가 좋아진다.
- 테스트를 작성하기 편해진다.
- 코드를 거의 수정하지 않고 데이터베이스를 손쉽게 변경할 수 있다.

1.1 SQL을 직접 다룰 때 발생하는 문제점

1.1.1 작업의 반복

- SQL을 직접 다루기 위해 자바와 관계형 데이터베이스를 사용해서 개발하면, 너무 많은 SQL 과 JDBC API 코드를 작성해야 한다.
- 테이블마다 이런 비슷한 일의 반복해야 하기 때문에 데이터 접근 계층(DAO)를 개발하는 일은 반복의 연속이다.



데이터 접근 계층(Data Access Object)

- 객체를 데이터베이스에 관리할 목적으로 만드는 오브젝트
- 즉 DB를 사용해 데이터를 조회하거나 조작하는 기능을 전담하도록 만든 오브젝트

1.1.2 SQL에 의존적인 개발

- 진정한 의미의 계층 분할이 어렵다.
 - 예를들어 Member 객체가 연관된 Team 객체를 사용할 수 있을지 없을지는 전적으로 사용하는 SQL에 달려있다.
 - 데이터 접근 계층을 사용해서 SQL을 숨겨도 어쩔 수 없이 DAO를 열어서 어떤 SQL이 실행되는지 확인해야 한다.
- 엔티티를 신뢰할 수 없다.
 - 비즈니스 요구사항을 모델링한 객체를 엔티티라 하는데, SQL에 모든 것을 의존하는 상황에서는 엔티티를 신뢰하고 사용할 수 없다.
- SQL에 의존적인 개발을 피하기 어렵다.
 - 물리적으로 SQL과 JDBC API를 데이터 접근 계층(DAO)에 숨기는데 성공했는지 몰라도 논리적으로 엔티티와 아주 강한 의존관계를 가지고 있다.
 - 이런 강한 의존관계 때문에 회원을 조회할 때는 물론 회원 객체에 필드를 하나 추가할 때도 DAO의 CRUD 코드와 SQL 대부분을 변경해야 하는 문제가 발생한다.

1.1.3 JPA와 문제 해결

- JPA를 사용하면 객체를 데이터베이스에 저장하고 관리할 때, 개발자가 직접 SQL을 작성하는 것이 아니라 JPA가 제공하는 API 를 사용하면 된다.
- 그러면 JPA가 적절한 SQL을 생성해서 데이터베이스에 전달한다.

1.2 패러다임의 불일치 40p

객체

- 객체는 속성(필드)과 기능(메소드)을 가진다.
- 객체의 기능은 클래스에 정의되어 있으므로 객체 인스턴스의 상태인 속성만 저장했다가 필요할 때 불러와서 복구하면 된다.
- 객체가 단순하면 객체의 모든 속성 값을 꺼내서 파일이나 데이터베이스에 저장하면 되지만, 부모 객체를 상속받았거나, 다른 객체를 참고하고 있다면 객체의 상태를 저장하기 쉽지 않다.

현실적인 대안은 관계형 데이터베이스에 객체를 저장하는 것이지만,

관계형 데이터베이스는 아래와 같은 특징을 가지고 있어서 **객체 구조를 테이블 구조에 저장하는 데 한계가 있다.**

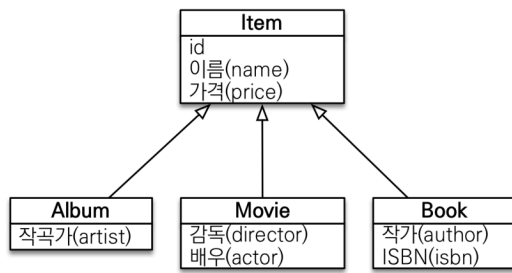
관계형 데이터 베이스

- 데이터 중심으로 구조화되어 있고 집합적인 사고를 요구한다.
- 추상화, 상속, 다형성 같은 개념이 없다.

⇒ 객체와 관계형 데이터베이스는 지향하는 목적이 서로 다르므로 둘의 기능과 표현 방법도 다른데, 이것을 객체와 관계형 데이터베이스의 **패러다임 불일치 문제**라고 한다.

패러다임의 불일치로 인해 발생하는 문제

1.2.1 상속 41p



객체 상속 관계

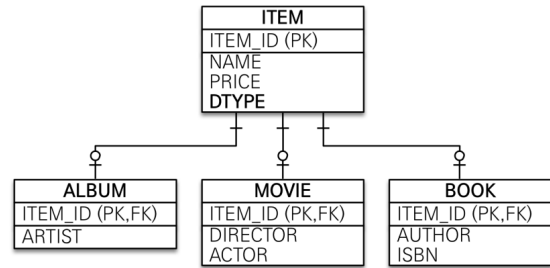


Table 슈퍼타입 서브타입 관계

- Album 객체를 저장하려면
 - 객체(Item, Album, Movie, Book)를 분해
 - SQL 문 만들기
 - JDBC API 를 사용하면,
 - 부모 객체에서 부모 데이터만 꺼내서 Item 용 INSERT SQL 작성
 - 자식 객체에서 자식 데이터만 꺼내서 Album 용 INSERT SQL 작성

⇒ 이 과정이 패러다임의 불일치를 해결하려고 소모하는 비용

- 해당 객체들을 데이터베이스가 아닌 자바 컬렉션에 보관한다면 부모, 자식 타입에 대한 고민 없이 컬렉션을 사용하면 됨.

```
// 저장
list.add(album);
list.add(movie);

// 조회
Album album = list.get(albumId);
```

JPA와 상속

```
// JPA를 사용해서 Item을 상속한 Album 객체를 저장
jpa.persist(album);
```

```
// 조회
Album album = jpa.find(Album.class, "id100");
```

1.2.2 연관관계 43p

객체

- 참조를 사용해서(`member.getTeam()`) 다른 객체와 연관관계를 가진다.
- 참조에 접근해서 연관된 객체를 조회한다.

테이블

- 외래 키를 사용해서(`JOIN ON M.TEAM_ID = T.TEAM_ID`) 다른 테이블과 연관관계를 가진다.
- 조인을 사용해서 연관된 테이블을 조회한다.

객체를 테이블에 맞추어 모델링을 하면

```
class Member {
    String id; // MEMBER_ID 컬럼 사용
    Long teamId; // TEAM_ID FK 컬럼 사용 /**
    String username; // USERNAME 컬럼 사용
}

class Team {
    Long id; // TEAM_ID PK 사용
    String name; // NAME 컬럼 사용
}
```

- 관계형 데이터베이스는 조인이라는 기능이 있으므로 외래 키의 값을 그대로 보관해도 되지만 객체는 연관된 객체의 참조를 보관해야 `Team team = member.getTeam();` 처럼 참조를 통해 연관된 객체를 찾을 수 있다.
- `Member.teamId` 필드처럼 TEAM_ID 외래 키까지 관계형 데이터베이스가 사용하는 방식에 맞추면 Member 객체와 연관된 Team 객체를 참조를 통해서 조회할 수 없다.

따라서 이런 방식을 따르면 좋은 객체 모델링은 기대하기 어렵고 결국 객체지향의 특징을 잃어버리게 된다.

객체지향 모델링

```
class Member {
    String id; // MEMBER_ID 컬럼 사용
    Team team; // 참조로 연관관계를 맺는다.
    String username; // USERNAME 컬럼 사용
}

class Team {
    Long id; // TEAM_ID PK 사용
    String name; // NAME 컬럼 사용
}
```

- 연관된 Team의 참조를 보관하여 `Team team = member.getTeam();` 으로 회원과 연관된 팀을 조회할 수 있다.
- 하지만 아래 이유 때문에 객체를 테이블에 저장하거나 조회하기가 쉽지 않고, 결국 개발자가 중간에서 변환 역할을 해줘야 한다. (직접 연관관계 설정하는 것은 47p 예제 참고)

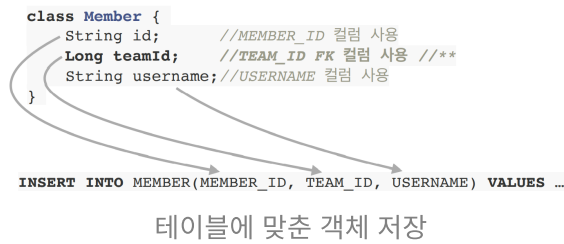
Member 객체

- team 필드로 연관관계를 맺는다.
- 객체 모델은 외래키가 필요 없고 단지 참조만 있으면 된다.

MEMBER 테이블

- TEAM_ID 외래키로 연관관계를 맺는다.
- 참조가 필요 없고 외래 키만 있으면 된다.

객체를 테이블에 맞추어 모델링, 객체지향 모델링 비교



JPA와 연관관계

- JPA는 연관관계와 관련된 패러다임의 불일치 문제를 해결해준다.

```

member.setTema(team); // 회원과 팀 연관관계 설정
jpa.persist(member);  // 회원과 연관관계 함께 저장
  
```

⇒ JPA는 team의 참조를 외래 키로 변환해서 적절한 INSERT SQL을 데이터베이스에 전달한다.

- JPA는 객체를 조회할 때 외래 키를 참조로 변환하는 일도 처리해준다.

```

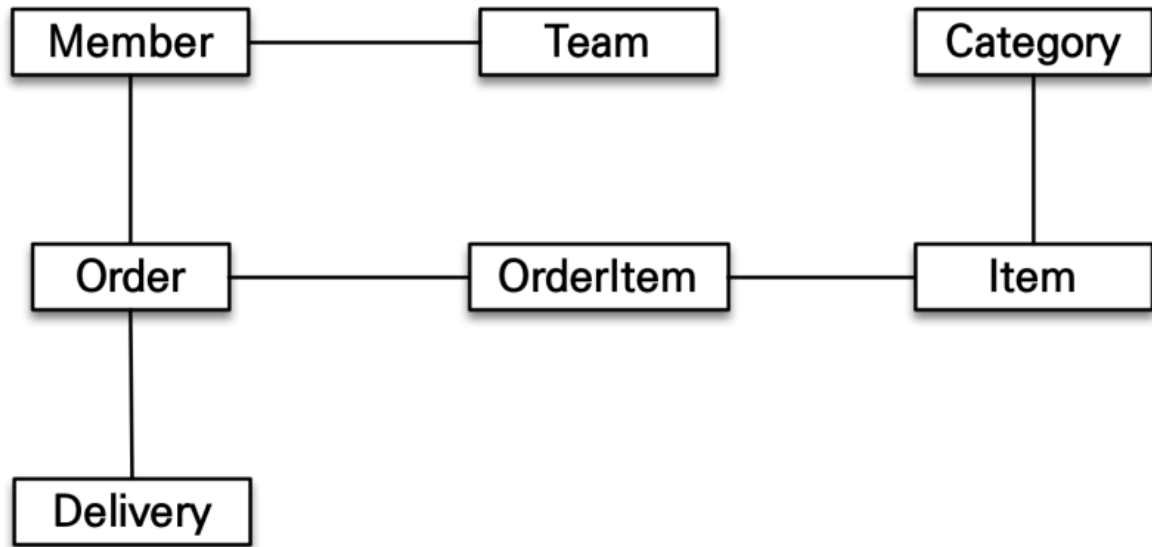
Member member = jpa.find(Member.class, memberId);
Team team = member.getTeam();
  
```

연관관계와 관련해서 극복하기 어려운 패러다임의 불일치 문제

1.2.3 객체 그래프 탐색 48p

객체 그래프 탐색

: 객체에서 회원(Member)이 소속된 팀을 조회할 때 `Team team = member.getTeam();` 처럼 참조를 사용해서 연관된 팀을 찾으면 되는데, 이것을 객체 그래프 탐색이라고 한다.



- 객체는 마음껏 객체 그래프를 탐색할 수 있어야 한다.

```
member.getOrder().getOrderItem()...
```

- SQL을 직접 다루면 처음 실행하는 SQL에 따라 객체 그래프를 어디까지 탐색할 수 있는지 정해진다.

```
SELECT M.*, T.*
FROM MEMBER M
JOIN TEAM T ON M.TEAM_ID = T.TEAM_ID;
```

SQL에서 이렇게 회원과 팀에 대한 데이터만 조회할 경우,

`member.getTeam()` 은 성공하지만, `member.getOrder();` 는 데이터가 없어서 null

⇒ 엔티티를 신뢰할 수가 없어진다.

DAO에 상황에 따라 조회할 수 있는 메서드를 여러개 만들어서 사용해야 하는데 모든 객체를 미리 로딩할 수는 없다.

JPA와 객체 그래프 탐색

- 객체 그래프를 마음껏 탐색할 수 있다. `member.getOrder().getOrderItem()...`
- JPA는 연관된 객체를 사용하는 시점에 적절한 SELECT SQL을 실행한다. 이 기능은 실제 객체를 사용하는 시점까지 데이터베이스 조회를 미룬다고 해서 **지연로딩**이라고 한다.

1.2.4 비교 51p

- 데이터베이스는 기본 키의 값으로 각 로우(row)를 구분
- 객체는 동일성(identity) 비교와 동등성(equality) 비교 두가지 방법이 있다.

동일성 비교

- == 비교
- 객체 인스턴스의 주소 값을 비교

동등성 비교

- equals() 메서드 사용
- 객체 내부의 값을 비교

따라서 테이블의 로우를 구분하는 방법과 객체를 구분하는 방법에 차이가 있다.

동일성(==) 비교

```
String memberId = "100";
Member member1 = memberDAO.getMember(memberId);
Member member2 = memberDAO.getMember(memberId);

member1 == member2; // false
```

```
class MemberDAO {
    //
    public Member getMember(String memberId) {
        String sql = "SELECT * FROM MEMBER WHERE MEMBER_ID = ?";
        ...
        //JDBC API, SQL 실행
        return new Member(...);
    }
}
```

member1과 member2는 같은 데이터베이스 로우에서 조회했지만,

객체 측면에서 볼 때 둘은 다른 인스턴스기 때문

(MemberDAO.getMember() 를 호출할 때마다 new Member()로 인스턴스가 새로 생성된다)

만약 객체를 컬렉션에 보관했다면

```
Member member1 = list.get(0);
Member member2 = list.get(0);

member1 == member2; // true
```

JPA와 비교

- JPA는 같은 트랜잭션일 때 같은 객체가 조회되는 것을 보장한다.

```
String memberId = "100";
Member member1 = memberDAO.getMember(memberId);
Member member2 = memberDAO.getMember(memberId);

member1 == member2; // true
```

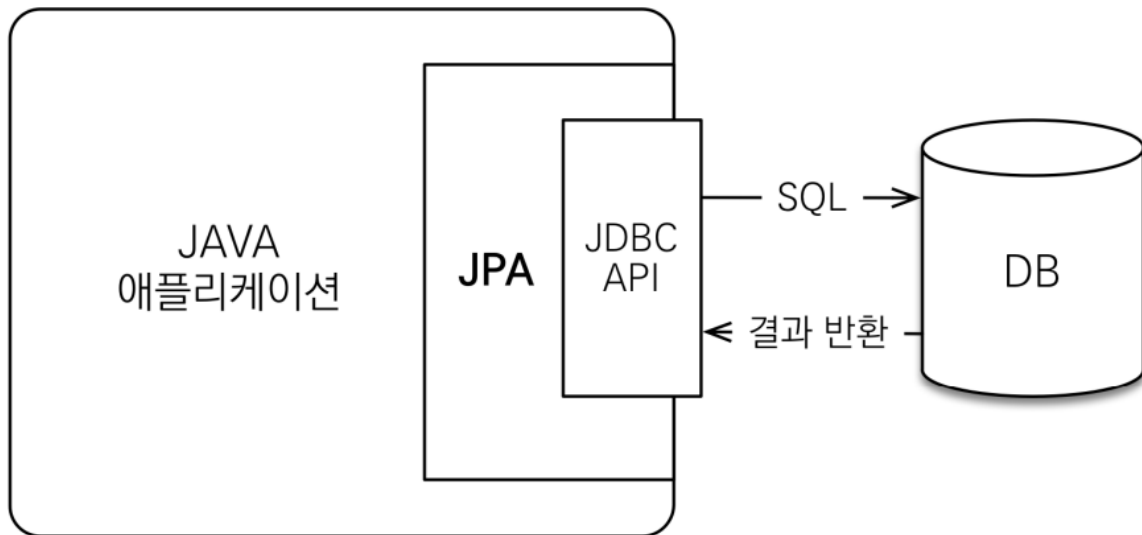
1.3 JPA(Java Persistence API)란? 54p

- 자바 진영의 ORM 기술 표준이다

ORM이란?

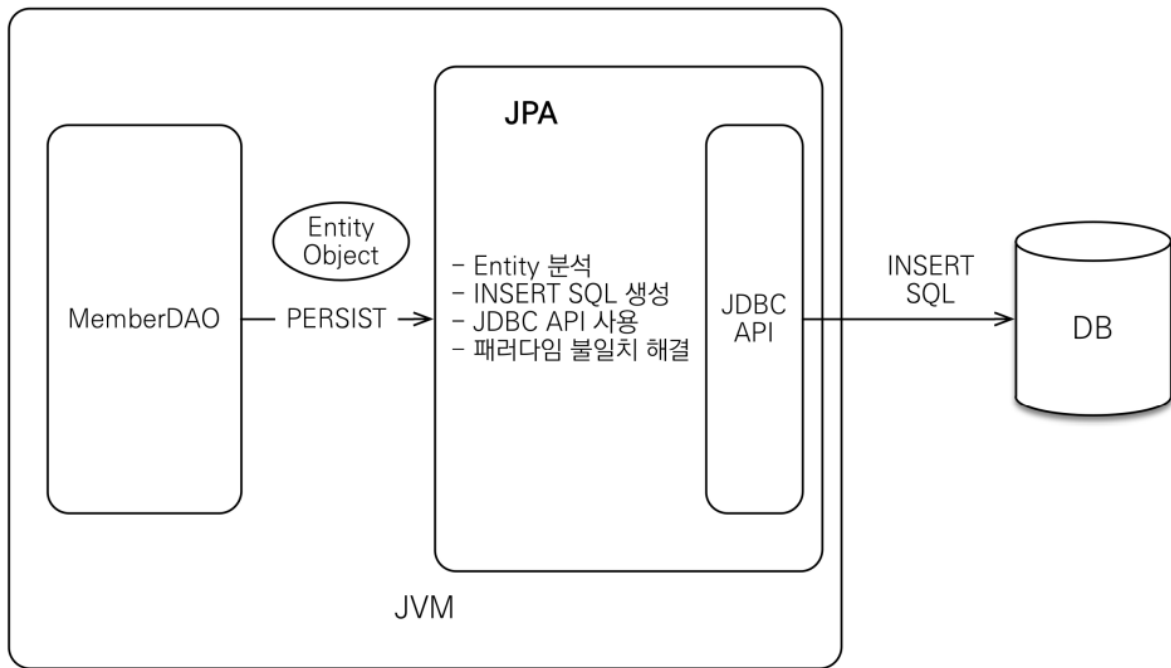
- Object-relational mapping(객체 관계 매핑)
- 객체와 관계형 데이터베이스를 매핑한다는 의미로 ORM 프레임워크가 중간에서 매핑한다.
- 객체를 데이터베이스에 저장할 때 직접 INSERT SQL을 작성하는 것이 아니라 객체를 마치 자바 컬렉션에 저장하듯이 ORM 프레임워크에 저장하면 된다.

- 대중적인 언어에는 대부분 ORM 기술이 존재한다.
- SQL을 대신 생성해서 데이터베이스에 전달해줄 뿐만 아니라 패러다임의 불일치 문제들도 해결해준다.
- 자바 진영에도 다양한 ORM 프레임워크들이 있는데 그중에 하이버네이트 프레임워크가 가장 많이 사용된다.



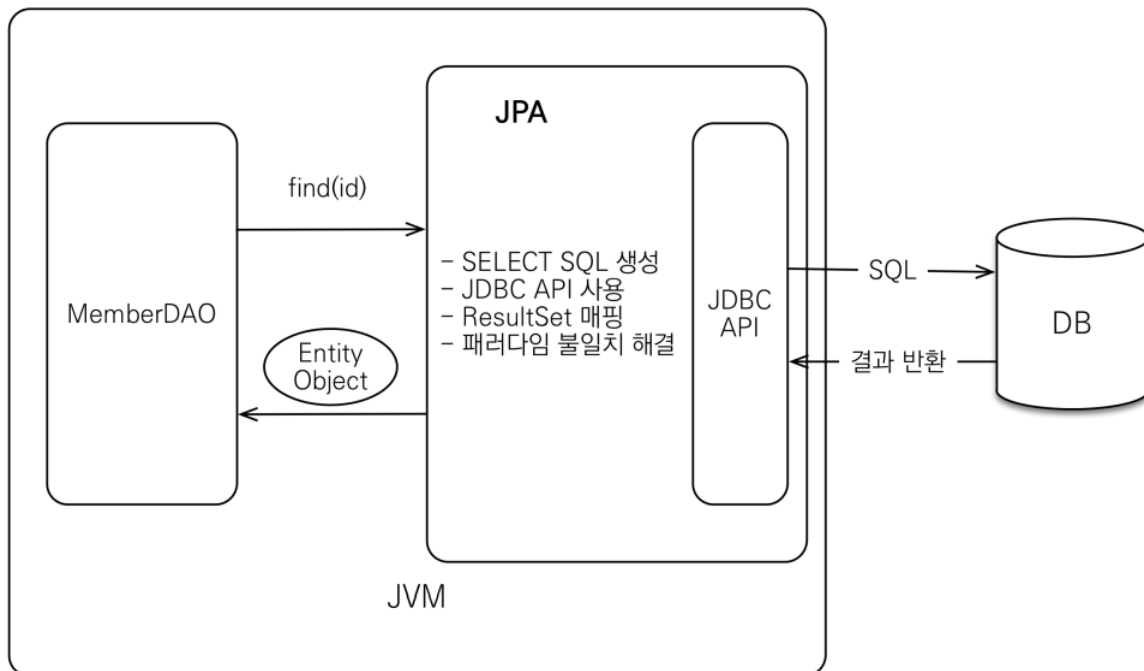
JPA는 애플리케이션과 JDBC 사이에서 동작

JPA 저장



JPA 동작 - 저장

JPA 조회

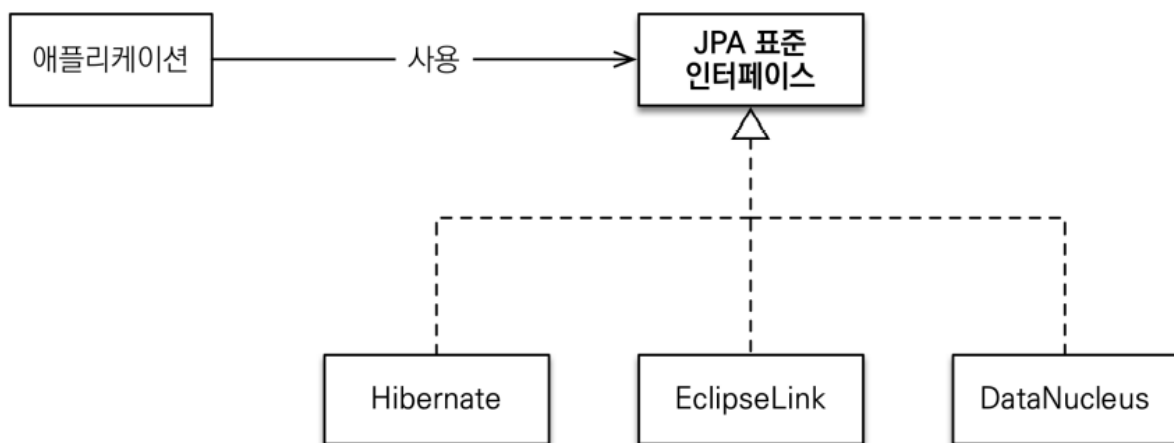


JPA 동작 - 조회

1.3.1 JPA 소개 56p

하이버네이트(오픈소스 ORM 프레임워크)를 기반으로 새로운 자바 ORM 기술 표준이 만들어졌는데 이것이 바로 JPA다.

- JPA는 자바 ORM 기술에 대한 API 표준 명세다.
- 즉 인터페이스를 모아둔 것.
- JPA를 사용하려면 JPA를 구현한 ORM 프레임워크를 선택해야 한다.
- JPA 2.1을 구현한 ORM 프레임워크는 하이버네이트, EclipseLink, DataNucleus 가 있는데 이 중에 하이버네이트가 가장 대중적이다.



1.3.2 왜 JPA를 사용해야 하는가? 57p

- 생산성
 - JPA를 사용하면 자바 컬렉션에 객체를 저장하듯이 JPA에 저장할 객체를 전달하면 된다.

```
jpa.persist(member); // 저장
Member member = jpa.find(memberId); // 조회
```

- 유지보수

- 필드를 추가하거나 삭제하면 개발자가 작성해야 했던 SQL과 JDBC API 코드를 JPA가 대신 처리해주므로 유지보수해야하는 코드 수가 줄어든다.
- 패러다임의 불일치를 해결해주므로 객체지향 언어가 가진 장점들을 활용해서 유연하고 유지보수하기 좋은 도메인 모델을 편리하게 설계할 수 있다.

- 패러다임의 불일치 해결

- ▼ 상속

- 개발자가 할 일 :

```
// 저장
jpa.persist(album);

// 조회
Album album = jpa.find(Album.class, albumId);
```

- 나머지 JPA가 처리 :

```
// 저장
INSERT INTO ITEM ...
INSERT INTO ALBUM...

// 조회
SELECT I.*, A.*
FROM ITEM I
JOIN ALBUM A ON I.ITEM_ID = A.ITEM_I
```

- ▼ 연관관계, 객체 그래프 탐색

```
// 연관관계 저장
member.setTeam(team);
jpa.persist(member);

// 객체 그래프 탐색
Member member = jpa.find(Member.class, memberId);
Team team = member.getTeam();
```

- ▼ 신뢰할 수 있는 엔티티, 계층

```
class MemberService {
    ...
    public void process() {
        Member member = memberDAO.find(memberId);
        member.getTeam(); //자유로운 객체 그래프 탐색
        member.getOrder().getDelivery();
    }
}
```

▼ 비교

```
String memberId = "100";
Member member1 = jpa.find(Member.class, memberId);
Member member2 = jpa.find(Member.class, memberId);
member1 == member2; // 같다.
```

⇒ 동일한 트랜잭션에서 조회한 엔티티는 같음을 보장

• 성능

1. 1차 캐시와 동일성(identity) 보장

- JPA는 애플리케이션과 데이터베이스 사이에서 동작한다.

```
String memberId = "helloId";
Member member1 = jpa.find(memberId); // SQL
Member member2 = jpa.find(memberId); // 캐시

member1 == member2 // true
```

- JPA를 사용하면 회원을 조회하는 SELECT SQL을 한 번만 데이터베이스에 전달하고 두 번째는 조회한 회원 객체를 재사용한다.

⇒ 같은 트랜잭션 안에서는 같은 엔티티를 반환 - 약간의 조회 성능 향상

- DB Isolation Level(격리 수준)이 Read Commit이어도 애플리케이션에서 Repeatable Read 보장

!! 격리 수준 관련 공부 필요

<https://blog.naver.com/sinjoker/222354471737>

2. 트랜잭션을 지원하는 쓰기 지연(transactional write-behind)

2.1 INSERT

- 트랜잭션을 커밋할 때까지 INSERT SQL을 모음
- JDBC BATCH SQL 기능을 사용해서 한번에 SQL 전송

```
transaction.begin(); // [트랜잭션] 시작
em.persist(memberA);
em.persist(memberB);
em.persist(memberC);
// 여기까지 INSERT SQL을 데이터베이스에 보내지 않는다.

// 커밋하는 순간 데이터베이스에 INSERT SQL을 모아서 보낸다.
transaction.commit(); // [트랜잭션] 커밋
```

2.2 UPDATE

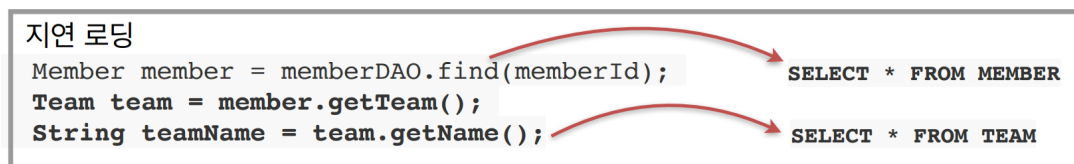
- UPDATE, DELETE로 인한 로우(ROW)락 시간 최소화
- 트랜잭션 커밋 시 UPDATE, DELETE SQL 실행하고, 바로 커밋

```
transaction.begin(); // [트랜잭션] 시작
changeMember(memberA);
deleteMember(memberB);
비즈니스_로직_수행(); // 비즈니스 로직 수행 동안 DB 로우 락이 걸리지 않는다.

// 커밋하는 순간 데이터베이스에 UPDATE, DELETE SQL을 보낸다.
transaction.commit(); // [트랜잭션] 커밋
```

3. 지연 로딩(Lazy Loading)

- 지연 로딩 : 객체가 실제 사용될 때 로딩




- 즉시 로딩 : JOIN SQL로 한번에 연관된 객체까지 미리 조회

즉시 로딩

```
Member member = memberDAO.find(memberId);  
Team team = member.getTeam();  
String teamName = team.getName();
```

```
SELECT M.*, T.*  
FROM MEMBER  
JOIN TEAM ...
```



- 데이터 접근 추상화와 벤더 독립성

- 애플리케이션과 데이터베이스 사이에 추상화된 데이터 접근 계층을 제공해서 애플리케이션이 특정 데이터베이스 기술에 종속되지 않도록 한다.