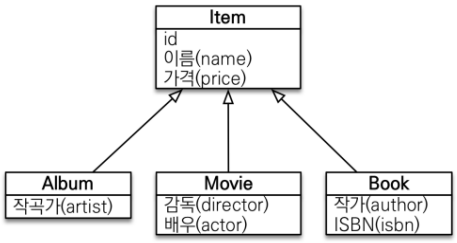
1. SQL 중심적인 개발의 문제점

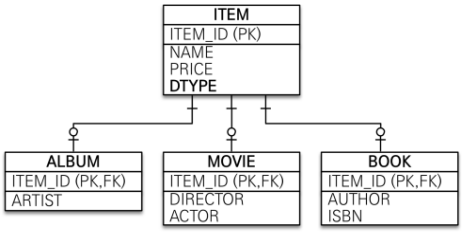
- 관계형 데이터베이스는 데이터 중심으로 구조화되어 있고, 집학적인 사고를 요구한다.
  그리고 객체지향에서 이야기하는 추상화, 상속, 다형성 같은 개념이 없다.
- 객체와 관계형 데이터베이스는 지향하는 목적이 서로 다르므로 둘의 기능과 표현 방법도 다르다.
  이것을 객체와 관계형 데이터베이스의 패러다임 불일치 문제라 한다.
- **문제는 객체와 관계형 데이터베이스 사이의 패러다임 불일치 문제를 해결하는데 너무 많은 시간과 코드를 소비하는 데 있다.**

1.1 상속

객체는 상속이라는 기능을 가지고 있지만 테이블은 상속이라는 기능이 없다. 그나마 데이터베이스 모델링에서 이야기하는 슈퍼타입 서브타입 관계를 사용하면 객체 상속과 가장 유사한 형태로 테이블을 설계할 수 있다.



[객체 상속 관계]



[Table 슈퍼타입 서브타입 관계]

1.2 JPA와 상속

JPA는 상속과 관련된 패러다임의 불일치 문제를 개발자 대신 해결해준다.
개발자는 마치 자바 컬렉션에 객체를 저장하듯이 JPA에게 객체를 저장하면 된다.

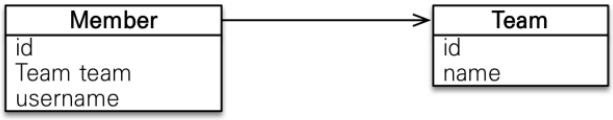
```
Album album = list.get(albumId);

// 부모 타입으로 조회 후 다형성 활용
Item item = list.get(albumId);
```

1.3 연관관계

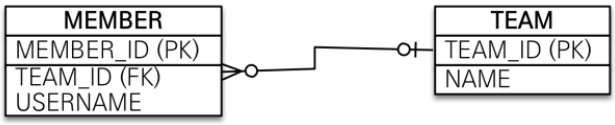
객체는 **참조**를 사용해서 다른 객체와 연관관계를 가지고 **참조**에 접근해서 연관된 객체를 조회한다.
반면에 테이블은 **외래 키**를 사용해서 다른 테이블과 연관관계를 가지고 조인을 해서 연관된 테이블을 조회한다.

- 객체는 참조를 사용: `member.getTeam()`
- 테이블은 외래 키를 사용: `JOIN ON M.TEAM_ID = T.TEAM_ID`

[객체 연관관계]



[테이블 연관관계]



1.4 객체를 테이블에 맞추어 모델링

```
class Member {
    String id; // MEMBER_ID(PK) 칼럼사용
    Long teamId;  // <- TEAM_ID(FK) 칼럼사용
    String username; // USERNAME 칼럼사용
}

class Team {
    Loing id; //TEAM_ID(PK) 사용
    String name; //NAME 칼럼사용
}
```

관계형 데이터베이스는 조인이라는 기능이 있기 때문에 외래 키의 값을 그대로 보관해도 된다.
하지만 객체는 연관된 객체의 참조를 보관해야한다. 다음처럼 구조를 통해 연관된 객체를 찾을 수 있다.

1.5 객체지향 모델링

```
class Member {
    String id; // MEMBER_ID(PK) 칼럼사용
    Team team; // TEAM_ID(FK) <- 참조로 연관관계를 맺는다.
    String username; // USERNAME 칼럼사용
}

class Team {
    Loing id; //TEAM_ID(PK) 사용
    String name; // NAME 칼럼사용
}
```

외래 키의 값을 그대로 보관하는 것이 아니라 연관된 Team의 참조를 보관한다.
그런데 이처럼 객체지향 모델링을 사용하면 객체를 테이블에 저장하거나 조회하기가 쉽지 않다.

- Member 객체는 team 필드로 연관관계를 맺고 Member 테이블은 TEAM_ID 외래 키로 연관관계를 맺기 때문이다.
- 반면에 테이블은 참조가 필요 없고 **외래 키**만 있으면 된다. 결국 개발자가 중간에 변화 역할을 해야 한다.

1.6 저장

- 객체를 데이터베이스에 저장하면 team 필드를 TEAM_ID 외래 키 값으로 변환 해야한다.
- 다음처럼 외래 키 값을 찾아서 INSERT SQL을 만들어야 한다.

```
member.getId(); //Member_ID (PK)에 저장
member.getTeam.getId(); // Team 객체의 Team_ID (PK)에 저장
member.getUsername(); // USERNAME 칼럼에 저장
```

1.7 객체 모델링 조회

- 조회할 때는 TEAM_ID 외래 키 값을 Member 객체의 team 참조로 변환해서 객체에 보관해야 한다.

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

public Member find(String memberId) {
    //SQL 실행
    Member member = new Member();
    ...

    //데이터베이스에 조회한 회원 관련 정보를 모두 입력
    Team team = new Team();

    member.setTeam(team);
    return member;
}
```

이러한 과정들은 모두 패러다임 불일치를 해결하려고 소모하는 비용이다.
만약 자바 컬렉션에 회원 객체를 저장한다면 이런 비용이 전혀들지 않는다.

2. JPA와 연관관계

JPA는 연관관계와 관련된 패러다임의 불일치 문제를 해결해준다.

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

- 개발자는 회원과 팀의 관계를 설정하고 회원 객체를 저장하면 된다.
- JPA는 team의 참조를 외래 키로 변환해서 적절한 INSERT SQL을 데이터베이스에 전달한다.
- 객체를 조회할 때 외래 키를 참조로 변환하는 일도 JPA가 처리해준다.

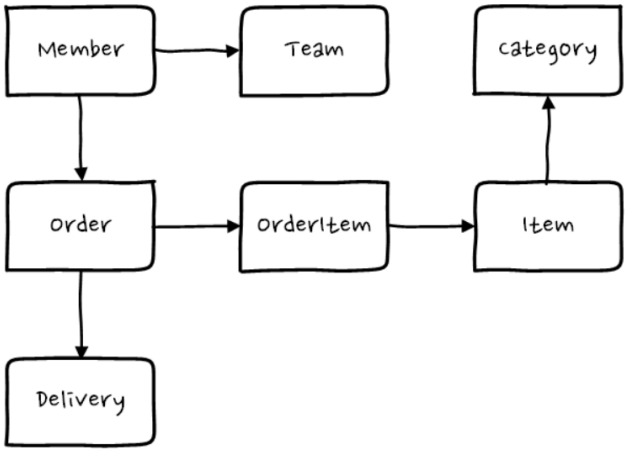
```
Member member = jpa.find(Member.class, memberId);
Team team = member.getTeam();
```

지금까지 설명한 문제들은 SQL을 직접 다뤄도 열심히 코드만 작성하면 어느정도 극복할 수 있는 문제다.
연관관계와 관련해서 극복하기 어려운 패러다임의 불일치 문제를 알아보자.

2.1 객체 그래프 탐색

객체에서 회원이 소속된 팀을 조회할 때는 밑의 그림처럼 참조를 사용해서 연관된 팀을 찾으면 되는데, 이것을 객체 그래프 탐색이라 한다.

```
Team team = member.getTeam();
```



객체는 마음껏 객체 그래프를 탐색할 수 있어야 한다.

```
member.getOrder().getOrderItem()... // 자유로운 객체 그래프 탐색
```

예를 들어 MemberDao에서 member 객체를 조회할 때,

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

위의 SQL을 실행해서 회원과 팀에 대한 데이터만 조회했다면 member.getTeam()은 성공하지만 다음처럼 다른 객체 그래프는 데이터가 없으므로 탐색할 수 없다.

```
member.getTeam(); //OK
member.getOrder(); //null
```

결국 SQL을 직접 다루면 처음 실행하는 SQL에 따라 객체 그래프를 어디까지 탐색할 수 있는지 정해진다. 이것은 객체지향 개발자에겐 너무 큰 제약이다. 왜냐하면 비즈니스 로직에 따라 사용하는 객체 그래프가 다른데, 언제 끊어질지 모를 객체 그래프를 함부로 탐색할 수는 없기 때문이다.

2.2 회원 조회 비즈니스 로직

```
class MemberService {
  ..
  public void process() {
    Member member = memberDao.find(memberId);
    member.getTeam(); // member->team 객체 그래프 탐색이 가능할까?
    member.getOrder().getDelivery(); // ??
  }
}
```

위 코드에서 memberDao를 통해서 member 객체를 조회했지만 이 코드만 보고는 Team, Order, Delivery 방향으로 객체 그래프 탐색을 할 수 있을지 없을지 전혀 예측할 수 없다. 결국 또 Dao를 열어서 SQL을 직접 확인해야 하는 것이다.

그렇다고 member와 연관된 모든 객체 그래프를 메모리에 올려두는 것은 현실성이 없다. 결국 memberDao에 회원을 조회 하는 메서드를 상황에 따라 여러개 만들어서 사용해야 한다.

```
memberDao.getMember();
memberDao.getMemberWithTeam();
memberDao.getMemberWithOrderWithDelivery();
```

그렇다면 JPA는 이 문제를 어떻게 해결하는지 보자.

2.3 JPA와 객체 그래프 탐색

JPA를 사용하면 객체 그래프를 마음껏 탐색할 수 있다.

앞에서 나왔듯이 JPA는 연관된 객체를 사용하는 시점에 적절한 SELECT SQL을 실행한다. 따라서 JPA를 사용하면 연관된 객체를 신뢰하고 조회할 수 있다. 이 기능은 실제 객체를 사용하는 시점까지 데이터베이스 조회를 미룬다고 해서 지연 로딩이라 한다.

JPA는 **지연 로딩을 투명(transparent)하게 처리**한다. 아래의 코드를 보면 메서드 구현 부분에 JPA에 관련된 어떤 코드도 직접 사용하지 않는다.

```
class Member {
  private Order order;

  public Order getOrder() {
    return order;
  }
  ...
```

2.4 지연 로딩 사용

```
// 처음 조회 시점에 SELECT MEMBER SQL
Member member = jpa.find(Member.class, memberId);

Order order = member.getOrder();
order.getOrderDate(); // Order를 사용하는 시점에 SELECT ORDER SQL
```

만약 Member를 사용할 때마다 Order를 사용한다면?

JPA는 연관된 객체를 즉시 함께 조회할지 아니면 실제 사용되는 시점에 지연해서 조회할지 간단한 설정으로 정의할 수 있다.

2.5 비교

데이터베이스는 기본 키의 값으로 각 로우(row)를 구분한다. 반면 **객체는 동일성(identity) 비교와 동등성(equality) 비교 두 방법**이 있다.

- 동일성 비교는 `==` 비교. **객체 인스턴스의 주소 값을 비교**한다.
- 동등성 비교는 `equals()` 메서드를 사용해서 **객체 내부의 값을 비교**한다.

따라서 테이블의 로우를 구분하는 방법과 객체를 구분하는 방법에는 차이가 있다.

2.6 MemberDao 코드

```
class MemberDao {

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

2.7 조회한 회원 비교하기

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

// member1 == member2 ==> false
```

member1과 member2는 같은 데이터베이스 로우에서 조회했지만, 객체 측면에선 다른 인스턴스이다. 따라서 객체의 동일성 비교에는 실패한다. 만약 객체를 컬렉션에 보관했다면 비교에 성공했을 것이다.

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

// `member1 == member2` ==> true
```

이런 패러다임 불일치 문제를 해결하기 위해 데이터베이스의 같은 로우를 조회할 때마다 같은 인스턴스를 반환하도록 구현하는 것은 쉽지 않다.

2.8 JPA와 비교

JPA는 같은 트랜잭션일 때 같은 객체가 조회되는 것을 보장한다. 그러므로 다음 코드에서 member1과 member2는 동일성 비교에 성공한다.

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

// member1 == member2 ==> true
```

객체 비교하기는 분산 환경이나 트랜잭션이 다른 상황까지 고려하면 더 복잡해진다. (책을 진행하면서 자세히)

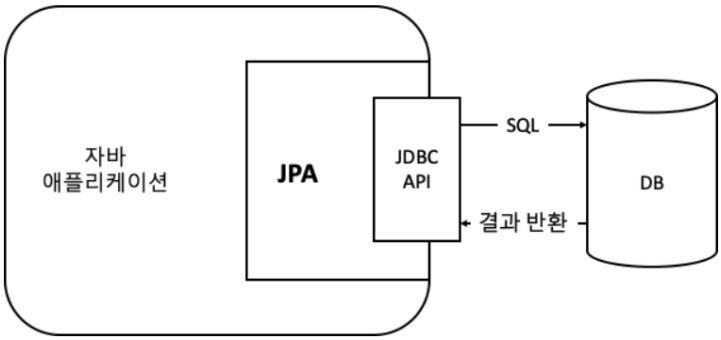
2.9 정리

객체 모델과 관계형 데이터베이스 모델은 지향하는 패러다임이 서로 다르다. 문제는 이 차이를 극복하려고 개발자가 너무 많은 시간과 코드를 소비한다는 점이다. 더 어려운 문제는 객체지향의 특성상 정교한 객체 모델링을 할수록 패러다임의 불일치 문제가 더 커진다는 것이다.

자바 진영에서는 패러다임의 불일치 문제를 해결하기 위해 많은 노력을 기울여왔다. 그리고 그 결과물이 JPA이다. JPA는 패러다임의 불일치 문제를 해결해주고 정교한 객체 모델링을 유지하게 도와준다. JPA를 문제 해결 위주로 간단히 살펴보았는데, 이제 본격적으로 JPA에 대해 알아보자.

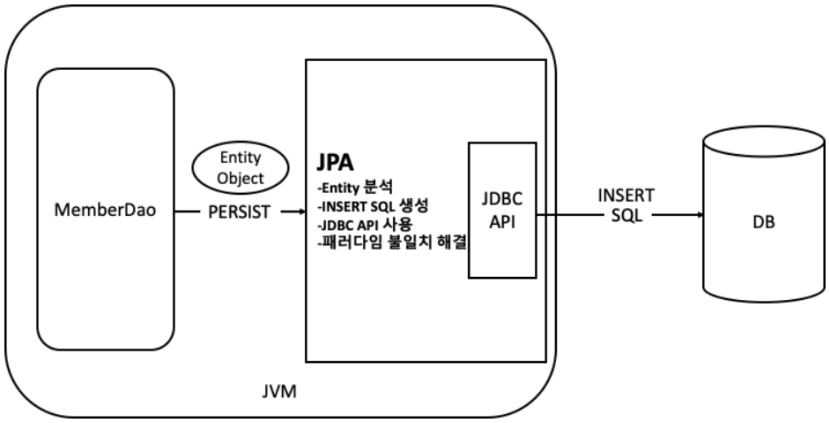
3. JPA란 무엇인가?

- JPA(Java Persistence API)는 자바 진영의 ORM 기술 표준
- JPA는 애플리케이션과 JDBC 사이에서 동작



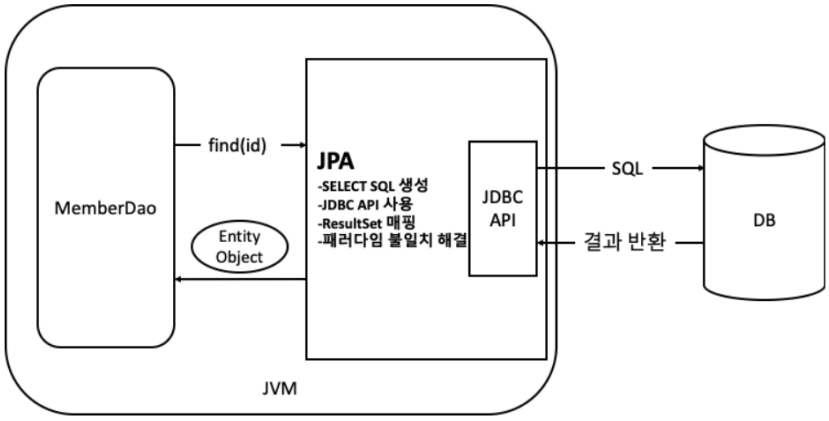
ORM(Object-Relational Mapping)은 이름 그대로 객체와 관계형 데이터베이스를 매핑한다는 뜻이다. ORM 프레임워크는 객체와 테이블을 매핑해서 패러다임의 불일치 문제를 개발자 대신 해결해준다. 예를 들어 객체를 데이터베이스에 저장할 때 INSERT SQL을 직접 작성하는 것이 아니라 객체를 마치 컬렉션에 저장하듯이 ORM 프레임워크에 저장하면 된다. 그러면 ORM 프레임워크가 적절한 INSERT SQL을 생성해서 데이터베이스에 객체를 저장해준다.

3.1 JPA 저장



조회할 때도 JPA를 통해 객체를 직접 조회하면 된다.

3.2 JPA 조회



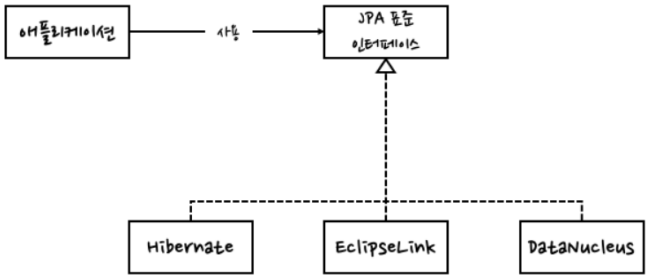
ORM 프레임워크는 단순히 SQL을 개발자 대신 생성해서 데이터베이스에 전달해주는 것뿐만 아니라 앞서 이야기한 다양한 패러다임의 불일치 문제들도 해결해주어 객체 측면에서는 정교한 객체 모델링을 할 수 있고 관계형 데이터베이스는 데이터베이스에 맞도록 모델링하면 된다. 덕분에 개발자는 데이터 중심인 관계형 데이터베이스를 사용해도 객체지향 애플리케이션에 집중할 수 있다.

3.3 JPA 소개

과거의 자바 진영에서는 EJB(Enterprise Java Beans)라는 기술 표준을 만들었는데 그 안에는 엔티티 빈이라는 ORM 기술도 포함되어 있었다. 하지만 너무 복잡하고 기술 성숙도도 떨어졌으며 자바 엔터프라이즈 애플리케이션 서버에서만 동작했다. 이때 하이버네이트(hibernate.org)라는 오픈소스 ORM 프레임워크가 등장했는데 EJB의 ORM 기술과 비교해서 가볍고 기술 성숙도도 높았다. 또한 자바 엔터프라이즈 애플리케이션 서버 없이도 동작했기때문에 많은 개발자들이 사용하기 시작했다.

결국 EJB 3.0에서 하이버네이트를 기반으로 새로운 자바 ORM 기술 표준이 만들어졌는데 이것이 바로 JPA이다.

3.4 JPA 표준 인터페이스와 구현체



JPA는 자바 ORM 기술에 대한 API 표준 명세다. 쉽게 말해서 인터페이스를 모아둔 것이다. 따라서 JPA를 사용하려면 JPA를 구현한 ORM 프레임워크를 선택해야 한다. 하이버네이트, EclipseLink, DataNucleus 중 하이버네이트가 가장 대중적이다.

JPA라는 표준 덕분에 특정 구현 기술에 대한 의존도를 줄일 수 있고 다른 구현 기술로 손쉽게 이동할 수 있다는 장점이 있다.

JPA 버전별 특징을 간략하게 정리하자면,

JPA 1.0(JSR 220) 2006년: 초기 버전이다. 복합 키와 연관관계 기능이 부족했다.
JPA 2.0(JSR 317) 2009년: 대부분의 ORM 기능을 포함하고 JPA Criteria가 추가되었다.
JPA 2.1(JSR 338) 2013년: 스토어드 프로시저 접근, 컨버터(Converter), 엔티티 그래프 기능이 추가되었다.

3.5 왜 JPA를 사용해야 하는가?

생산성

JPA를 사용하면 컬렉션에 객체를 저장하듯이 JPA에게 저장할 객체를 전달하면 된다. 지루하고 반복적인 일은 JPA가 대신 처리해준다.

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

더 나아가 JPA에는 CREATE TABLE같은 DDL 문을 자동으로 생성해주는 기능도 있다. 이런 기능들을 사용하면 데이터베이스 설계 중심의 패러다임을 객체 설계 중심으로 역전시킬 수 있다.

유지보수

SQL을 직접 다루면 엔티티에 필드 하나만 추가해도 관련된 코드들을 모두 변경해야 했다. 반면에 JPA는 이런 과정을 대신 처리해주므로 필드를 추가하거나 삭제해도 수정해야 할 코드가 줄어든다. 또한 패러다임 불일치 문제를 해결해준다. 객체지향 언어가 가진 장점들을 활용하여 유지보수하기 좋은 도메인 모델을 편리하게 설계할 수 있다.

패러다임의 불일치 해결

JPA는 상속, 연관관계, 객체 그래프 탐색, 비교하기와 같은 패러다임의 불일치 문제를 해결해준다.

성능

JPA는 애플리케이션과 데이터베이스 사이에서 다양한 성능 최적화 기회를 제공한다.
JPA는 애플리케이션과 데이터베이스 사이에서 동작하므로 최적화 관점에서 시도해볼 수 있는 것들이 많다.

```
String memberId = "helloId";
Member member1 = jpa.find(memberId);
Meber member2 = jpa.find(memberId);
```

JDBC를 사용해서 해당 코드를 직접 작성했다면 회원을 조회할 때마다 SELECT SQL을 사용해서 데이터베이스와 두 번 통신했을 것이다. JPA는 한 번만 데이터베이스에 전달하고 두 번째부터는 조회한 객체를 재사용한다.

데이터 접근 추상화와 벤더 독립성

관계형 데이터베이스는 같은 기능도 벤더마다 사용법이 다른 경우가 많다. 애플리케이션은 처음 선택한 데이터베이스 기술에 종속되고 다른 데이터베이스로 변경하기는 매우 어렵다.

JPA는 애플리케이션과 데이터베이스 사이에 추상화된 데이터 접근 계층을 제공해서 특정 데이터베이스 기술에 종속되지 않도록 한다.
만약 데이터베이스를 변경하면 JPA에게 다른 데이터베이스를 사용한다고 알려주기만 하면 된다.

표준

JPA는 자바 진영의 ORM 표준이다. 앞서 이야기 했듯이 표준을 사용하면 다른 구현 기술로 손쉽게 변경할 수 있다.

정리
지금까지 SQL을 직접 다룰 때 발생하는 다양한 문제와 객체지향 언어와 관계형 데이터베이스 사이의 패러다임 불일치 문제를 설명했다. 그리고 JPA가 각 문제를 어떻게 해결하는지 알아보았다. JPA에 관한 자세한 내용은 차근차근 살펴보기로 하고, 우선은 다음 장부터 테이블 하나를 등록 / 수정 / 삭제 / 조회하는 간단한 JPA 애플리케이션을 만들어보자.