**CHATER1**

* 1. **SQL을 직접 다룰 때 발생하는 문제점**

**1.1.1 반복, 반복 그리고 반복**

-데이터베이스는 객체 구조와는 다른 데이터 중심의 구조를 가지므로 객체를 데이터베이스에 직접 저장허간 조회할 수 없다.

-중간에서 SQL과 JDBC API를 사용해서 변환 작업

🡺너무 많은 SQL과 JDBC API를 코드로 작성해야 된다.

**1.1.2 SQL에 의존적인 개발**

-필드를 추가하고, SQL 쿼리도 수정해야 된다.

-만약 자바 컬렉션이라면

ex)

list.add(member) // 등록

Member member = list.get(xxx); // 조회

Member.setTel(“xxx”); // 수정

-Member나 Team처럼 비즈니스 요구사항을 모델링한 객체를 엔티티 라고 한다.

-애플리케이션에서 SQL을 직접 다룰 때 문제점

1) 진정한 의미의 계층 분할이 어렵다.

2) 엔티티를 신뢰할 수 없다. (계속 변경)

3) SQL에 의존적인 개발을 피하기 어렵다.

**1.1.3 JPA와 문제 해결**

-JPA를 사용하면 SQL을 직접 작성하는 것이 아니라 JPA가 제공하는 API를 사용하면 된다.

그러면 JPA가 개발자 대신에 적절한 SQL을 생성해서 데이터베이스에 전달한다.

**1.2 패러다임의 불일치**

-객체지향 프로그래밍은 추상화, 캡슐화, 정보은닉, 상속, 다형성 등 복잡성을 제어할 수 있는 다양한 장치들을 제공한다.

-도메인 모델도 객체로 모델링하면 객체지향 언어가 가진 장점들을 활용할 수 있다.

-하지만 부모 객체를 상속받았거나, 다른 객체를 참조하고 있다면 객체의 상태를 저장하기 쉽지 않다.

**🡺그러면 관계형은?** 데이터 중심으로 구조화되어 있기 때문에 객체지향에서 이야기하는 추상화, 상속, 다형성 같은 개념이 없다.

**1.2.1 상속**

-테이블은 상속이라는 기능이 없다. 그나마 슈퍼타입 서브타입 관계를 사용하면 유사한 형태로 테이블을 설계할 수 있다.

**JPA와 상속**

-JPA는 상속과 관련된 패러다임의 불일치 문제를 개발자 대신 해결해준다. 개발자는 컬렉션에 객체를 저장하듯이JPA에게 객체를 저장하면 된다.

ex)

**저 장**

jpa.persist(album);

🡺INSERT INTO ITEM …

INSERT INTO ALBUM …

**조 회**

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

🡺 SELECT I.\*, A.\*

FROM ITEM i

JOIN ALBUM A ON I.ITEM\_ID = A.ITEM\_ID

**1.2.2 연관관계**

-객체는 참조를 사용해서 다른 객체와 연관관계를 가지고 참조에 접근해서 연관된 객체를 조회한다.

-반면에 테이블은 외래 키를 사용해서 다른 테이블과 연관관계를 가지고 조인을 사용해서 연관된 테이블을 조회한다.

-객체는 참조가 있는 방향으로만 조회할 수 있는 반면, 테이블은 외래 키 하나로 반대로도 조회 가능하다.

**잘못된 방법**

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  
}

🡺Member 객체와 연관된 Team 객체를 참조를 통해서 조회할 수 없다.

**정상적인 방법**

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  
}

🡺객체를 테이블에 저장하거나 조회하기 쉽지 않다. (테이블은 외래키가 있어야 된다.)

**JPA와 연관관계**

-JPA는 연관관계와 관련된 패러다임의 불일치 문제를 해결해준다.

🡺member.setTeam(team); // 회원과 팀 연관관계 설정

jpa.persist(member); // 회원과 연관관계 함께 저장

**1.2.3 객체 그래프 탐색**

-참조를 사용해서 연관된 객체를 찾는 것

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

-객체 그래프 탐색은 코드만 보고 어디까지 할 수 있는지 예측할 수 없다. DAO를 열어서 SQL을 직접 확인해야 한다. (엔티티가 SQL 종속)

🡺연관된 모든 객체 그래프를 조회하면? 메모리 부담

**JPA와 객체 그래프 탐색**

-JPA를 사용하면 **지연 로딩**을 사용해서 신뢰된 객체를 마음껏 조회 가능

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

🡺실제 객체를 불러 올 때 자주 사용되나 여부에 따라 PreLoading, LazyLoading 전략을 취한다.

**1.2.4 비교**

-동일성 비교 (==) : 인스턴스의 주소 값 비교

-동등성 비교 (equals) : 객체 내부의 값 비교

-SQL 조회

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

🡺컬렉션이라면? 같았을텐데… 패러다임의 불일치

-JPA 조회

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

🡺동일하다.

**1.3 JPA란 무엇인가?**

-JPA는 자바 진영의 ORM 기술 표준



ORM은?

-객체와 관계형 데이터베이스 매핑

-ORM은 단순히 SQL을 개발자 대신 생성해서 데이터베이스에 전달할 뿐만 아니라 다양한 패러다임의 불일치까지 해결해준다.

**1.3.1 JPA 소개**

-EJB3.0에서 하이버네이트를 기반으로 새로운 자바 ORM 기술 표준이 만들어졌는데 이것이 JPA다.

-간략 버전 특징

1) JPA 1.0 (2006년) : 초기 버전. 복합 키와 연관관계 기능 부족

2) JPA 2.0 (2009년) : 대부분의 ORM 기능 포함. JPA Criteria가 추가됨

3) JPA 3.0 (2013년) : SP, 컨버터, 엔티티 그래프 기능등이 추가

**1.3.2 왜 JPA를 사용해야 하는가?**

생산성

-자바 컬렉션에 객체를 저장하듯이 JPA에게 저장할 객체를 전달하면 된다.

-지루하고 반복적인 코드와 CRUD용 SQL을 개발자가 직접 작성하지 않아도 된다.

유지보수

-번거로운 부분을 JPA가 대신 처리해주므로 수정해야 할 코드가 줄어든다.

-페러다임의 불일치도 해결해주므로 유지보수하기 좋은 도메인 모델을 편리하게 설계가능

패러다임의 불일치 해결

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

성능

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

-트랜잭션 안에서 두번째 조회 시, 첫번째 조회결과를 재사용한다.

데이터 접근 추상화와 벤더 독립성

-관계형 데이터 베이스는 벤더마다 사용법이 다른 경우가 많다.

🡺기술에 종속됨

-JPA는 추상화 계층을 제공해서 데이터베이스 기술에 종속되지 않도록 한다.

🡺연결 정보만 바꾸면 된다.

**CHAPTER2**

**2.3 라이브러리와 프로젝트 구조**

-JPA 구현체로 하이버네이트를 사용하기 위한 핵심 라이브러리

* hibernate-core : 하이버네이트 라이브러리
* hibernate-entitymanager : 하이버네이트가 JPA 구현체로 동작하도록 JPA 표준을 구현한 라이브러리
* hibernate-jpa-2.1-api : JPA 2.1 표준 API를 모아둔 라이브러리

**2.3.1 메이븐과 사용 라이브러리 관리**

-메이븐은 라이브러리 관리 도구로서 pom.xml을 사용한다.

<dependencies>  
 <!-- JPA, 하이버네이트 -->  
 <dependency>  
 <groupId>org.hibernate</groupId>  
 <artifactId>hibernate-entitymanager</artifactId>  
 <version>${hibernate.version}</version>  
 </dependency>  
 <!-- H2 데이터베이스 -->  
 <dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <version>${h2db.version}</version>  
 </dependency>  
</dependencies>

-hibernate-entitymanager : JPA 표준과 하이버네이트를 포함.

* hibernate-core.jar
* hibernate-jpa-2.1.-api.jar

-h2 데이터베이스 : h2 데이터베이스에 접속

**2.4 객체 매핑 시작**

회원 테이블

**CREATE TABLE** MEMBER (  
 ID **VARCHAR**(255) **NOT NULL**, --아이디(기본 키)  
 **NAME VARCHAR**(255), --이름  
 AGE **INTEGER NOT NULL**, --나이  
 **PRIMARY KEY** (ID)  
)

회원 클래스

@Entity  
@Table(name = "MEMBER")  
public class Member {  
  
 @Id  
 @Column(name = "ID")  
 private String id;  
  
 @Column(name = "NAME")  
 private String username;  
  
 private Integer age;  
  
 public String getId() {  
 return id;  
 }  
 public void setId(String id) {  
 this.id = id;  
 }  
 public String getUsername() {  
 return username;  
 }  
 public void setUsername(String username) {  
 this.username = username;  
 }  
 public Integer getAge() {  
 return age;  
 }  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
}

-@Entity : 테이블과 매핑한다고 JPA에 알려준다. @Entity가 사용된 클래스를 엔티티 클래스라고 한다.

-@Table : 엔티티 클래스에 매핑할 테이블 정보를 알려준다. 생략하면 클래스 이름과 매핑

-@Id : 엔티티 클래스의 필드를 테이블의 기본 키(Primary key)에 매핑한다. (식별자 필드)

-@Column : 필드를 컬럼에 매핑한다.

-매핑 정보가 없는 필드 : 필드명을 사용해서 컬럼명으로 매핑한다. (대소문자 구분할 경우 명시적으로 @Column(name=”AGE”) )

**2.5 persistence.xml 설정**

META-INF/persistence.xml 클래스 패스 경로에 있으면 별도의 설정 없이 JPA가 인식한다.

(Spring에서는 root-context.xml에 설정)

<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">  
 <persistence-unit name="jpabook">  
 <properties>  
 <!-- 필수 속성 -->  
 <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>  
 <property name="javax.persistence.jdbc.user" value="sa"/>  
 <property name="javax.persistence.jdbc.password" value=""/>  
 <property name="javax.persistence.jdbc.url" value="jdbc:h2:tcp://localhost/~/test"/>  
 <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />  
  
 <!-- 옵션 -->  
 <property name="hibernate.show\_sql" value="true" />  
 <property name="hibernate.format\_sql" value="true" />  
 <property name="hibernate.use\_sql\_comments" value="true" />  
 <property name="hibernate.id.new\_generator\_mappings" value="true" />  
  
 <!--<property name="hibernate.hbm2ddl.auto" value="create" />-->  
 </properties>  
 </persistence-unit>  
</persistence>

JPA 표준 속성

-javax.persistence.jdbc.driver : JDBC 드라이버

-javax.persistence.jdbc.user : 데이터베이스 접속 아이디

-javax.persistence.jdbc.password : 데이터베이스 접속 비밀번호

-javax.persistence.jdbc.url : 데이터베이스 접속 URL

하이버네이트 속성

-hibernate.dialect: 데이터베이스 설정

**2.5.1 데이터베이스 방언**

-데이터베이스 방언 : 데이터베이스 고유 기능

🡺하이버네이트를 포함한 대부분 JPA 구현체는 이런 문제를 해결하고자 데이터베이스 방언 클래스를 제공한다.

<https://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single/#configuration-optional-dialects>

<property name="hibernate.show\_sql" value="true" />  
<property name="hibernate.format\_sql" value="true" />  
<property name="hibernate.use\_sql\_comments" value="true" />  
<property name="hibernate.id.new\_generator\_mappings" value="true" />

-hibernate.show.sql : 하이너베이트가 실행한 SQL을 출력한다.

-hibernate.format\_sql : 하이버네이트가 실행한 SQL을 출력할 때 보기 쉽게 정렬한다.

-hibernate.use\_sql\_comments : 쿼리를 출력할 때 주석도 함께 출력한다.

-hibernate.id.new\_generator\_mappings : JPA 표준에 맞춘 새로운 키 생성 전략을 사용한다.

**2.6 애플리케이션 개발**

public static void main(String[] args) {  
  
 //엔티티 매니저 팩토리 생성  
 EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("jpabook");  
 EntityManager em = emf.createEntityManager(); //엔티티 매니저 생성  
 EntityTransaction tx = em.getTransaction(); //트랜잭션 기능 획득  
  
 try {  
 tx.begin(); //트랜잭션 시작  
 *logic*(em); //비즈니스 로직  
 tx.commit();//트랜잭션 커밋  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 tx.rollback(); //트랜잭션 롤백  
 } finally {  
 em.close(); //엔티티 매니저 종료  
 }  
  
 emf.close(); //엔티티 매니저 팩토리 종료  
}

크게 3개로 나누어져 있다.

1) 엔티티 매니저 설정

2) 트랜잭션 관리

3) 비즈니스 로직

**2.6.1 엔티티 매니저 설정**

엔티티 매니저 팩토리 생성

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("jpabook");

-persistence.xml의 설정 정보를 사용해서 엔티티 매니저 팩토리 생성

-엔티티 매니저 팩토리는 비용이 비싸므로 한 번만 생성하고 공유해서 사용한다.

엔티티 매니저 생성

EntityManager em = emf.createEntityManager();

-JPA의 기능 대부분은 엔티티 매니저가 제공

-엔티티를 데이터베이스에 등록/수정/삭제/조회할 수 있다.

-내부에 데이터소스를 유지하면서 데이터베이스와 통신한다. (애플리케이션 입장에서는 가상의 데이터베이스)

-데이터베이스 커넥션과 밀접한 관계가 있으므로 스레드간에 공유하거나 재사용하면 안 된다.

**2.6.2 트랜잭션 관리**

-JPA를 사용할 경우 항상 트랜잭션 안에서 데이터를 변경해야 한다.

EntityTransaction tx = em.getTransaction(); //트랜잭션 기능 획득  
  
try {  
 tx.begin(); //트랜잭션 시작  
 *logic*(em); //비즈니스 로직  
 tx.commit();//트랜잭션 커밋  
  
} catch (Exception e) {  
 e.printStackTrace();  
 tx.rollback(); //트랜잭션 롤백  
}

-로직이 정상 동작하면 commit하고 예외가 발생하면 rollback 한다.

**2.6.3 비즈니스 로직**

public static void logic(EntityManager em) {  
 //등록  
 em.persist(member);  
  
 //수정  
 member.setAge(20);  
  
 //한 건 조회  
 Member findMember = em.find(Member.class, id);  
  
 //목록 조회  
 List<Member> members = em.createQuery("select m from Member m", Member.class).getResultList();  
  
 //삭제  
 em.remove(member);

등록

-엔티티를 저장하려면 persist() 메소드에 저장할 엔티티를 넘겨주면 된다.

수정

-update문이 없고 도메인에서 set을 하면 트랜잭션 범위내에서 자동으로 저장된다.

삭제

-remove() 메소드를 이용해서 삭제

한 건 조회

-@Id로 데이터베이스 테이블의 기본 키와 매핑한 식별자 값으로 조회 한다.

**2.6.4 JPQL**

-JPA는 SQL을 추상화한 JPQL이라는 객체지향 쿼리 언어를 제공한다.

-SQL과 JPQL의 차이점

* JPQL은 엔티티 객체를 대상으로 쿼리한다. (클래스와 필드를 대상으로 쿼리)
* SQL은 데이터베이스 테이블을 대상으로 쿼리한다.

em.createQuery("select m from Member m", Member.class).getResultList();

-여기서 Member는 테이블이 아니라 엔티티 객체를 말하는것이다. JPQL은 데이터베이스 테이블을 전혀 알지 못한다.

🡺JPA가 JPQL을 분석 후, SQL을 만들어서 데이터베이스에서 데이터를 조회한다.

**CHAPTER3**

**3.1 엔티티 매니저 팩토리와 엔티티 매니저**

-엔티티 매니저는 데이터베이스 연결이 꼭 필요한 시점까지 커넥션을 얻지 않는다. (트랜잭션을 시작할 때 획득)

**3.2 영속성 컨텍스트란? (엔티티를 영구 저장하는 환경)**

-em.persist() : 엔티티 매니저를 사용해서 회원 엔티티를 영속성 컨텍스트에 저장한다.

-영속성 컨텍스트는 엔티티 매니저를 생성할 때 하나 만들어 진다.

**3.3 엔티티의 생명주기**

-엔티티에는 4가지 상태 존재

* 비영속(new/transient) : 영속성 컨텍스트와 전혀 관계가 없는 상태
* 영속(managed) : 영속성 컨텍스트에 저장된 상태
* 준영속(detached) : 영속성 컨텍스트에 저장되었다가 분리된 상태
* 삭제(removed) : 삭제된 상태

**Lifecycle**



**비영속**

-엔티티 객체를 생성했다. 순수한 객체 상태이며 아직 저장하지 않았다.

Member member = new Member();  
member.setId(id);  
member.setUsername("지한");

**영속**

-엔티티 매니저를 통해서 엔티티를 영속성 컨텍스트에 저장했다.

-영속성 컨텍스트가 관리하는 엔티티를 영속 상태라고 한다.

em.persist(member);

**준영속**

-영속 상태의 엔티티를 영속성 컨텍스트가 관리하지 않으면 준영속 상태가 된다.

em.detach(member);

**삭제**

-엔티티를 영속성 컨텍스트와 데이터베이스에서 삭제한다.

em.remove(member);

**3.4. 영속성 컨텍스트의 특징**

**영속성 컨텍스트와 식별자 값**

-영속 상태는 식별자 값이 반드시 있어야 한다.

**영속성 컨텍스트와 데이터베이스 저장**

-트랜잭션을 커밋하는 순간 데이터베이스에 반영하는데 이것을 플러시(flush)라 한다.

**영속성 컨텍스트가 엔티티를 관리시 장점**

-1차캐시, 동일성 보장, 쓰기 지연, 변경 감지, 지연 로딩 등

**3.4.1 엔티티 조회**

-영속성 컨텍스트는 내부에 캐시를 가지고 있는데(1차 캐시) 영속 상태의 엔티티는 모두 이곳에 저장

-em.find()를 호출 하면 먼저 1차 캐시에서 엔티티를 찾고 없으면 데이터베이스 조회

**1차 캐시에서 조회**

Member member = new Member();  
member.setId(id);  
member.setUsername("지한");

// 1차 캐시에 저장됨  
em.persist(member);  
  
// 1차 캐시에서 조회

Member findMember = em.find(Member.class, id);

**데이터베이스에서 조회**

-1차 캐시에 없으면 엔티티 매니저는 데이터베이스를 조회. 그리고 1차 캐시에 저장한 후에 영속 상태의 엔티티를 반환. (성능상 이점)

**영속 엔티티의 동일성 보장**

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

**3.4.2 엔티티 등록**

-트랜잭션을 커밋하기 직전까지 데이터베이스에 엔티티를 저장하지 않고 내부 쿼리 저장소

(쓰기 지연 SQL 저장소)에 데이터를 모아 놓는다. 그리고 트랜잭션을 커밋할 때 모아둔 쿼리를 데이터베이스로 보낸다.

🡺트랜잭션을 지원하는 **쓰기 지연** 이라고 한다.

-트랜잭션 커밋하면 쓰기 지연 SQL 저장소에 모인 쿼리를 데이터베이스에 보낸다.

**3.4.3 엔티티 수정**

**SQL 수정 쿼리의 문제점**

-수정 쿼리가 많아지는 것은 물론이고 비즈니스 로직을 분석하기 위해 SQL을 계속 확인해야 한다.

**변경감지 (JPA는?)**

transaction.begin(); //트랜잭션 시작  
  
// 영속 엔티티 조회  
Member memberA = em.find(Member.class, "memberA");  
  
// 영속 엔티티 데이터 수정  
memberA.setUsername("hi");  
memberA.setAge(10);  
  
transaction.commit();//트랜잭션 커밋

-단순히 엔티티를 조회해서 데이터만 변경하면 된다.

-이렇게 엔티티의 변경사항을 데이터베이스에 자동으로 반영하는 기능을 **변경감지** 라 한다.

-JPA는 엔티티를 영속성 컨텍스트에 보관할 때, 최초 상태를 복사해서 저장해두는데 이것을 스냅샷이라고 한다. 그리고 플러시 시점에 스냅샷과 엔티티를 비교해서 변경된 엔티티를 찾는다.

-변경 감지는 영속 상태의 엔티티에만 적용된다.

**JPA의 기본 전략은 엔티티의 모든 필드를 업데이트한다. (장점)**

-모든 필드를 사용하면 수정 쿼리가 항상 같다. (재사용)

-데이터베이스에 동일 쿼리를 보내면 이전에 파싱된 쿼리를 재사용할 수 있다.

-필드가 많거나 저장되는 내용이 크면 수정된 데이터만 사용해서 동적으로 Update 하면 된다.

(@DynamicUpdate)

@Entity  
@DynamicUpdate  
@Table(name = "MEMBER")  
public class Member {

. . .

}

**3.4.4 엔티티 삭제**

-em.remove를 호출하는 순간 영속성 컨텍스트에서 제거된다.

**3.5 플러시**

-플러시(flush)는 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영한다.

-영속성 컨텍스트는 비우지 않는다.

**플러시 하는 3가지 방법**

1) em.flush() 직접 호출

2) 트랜잭션 커밋 시 플러시가 자동 호출

3) JPQL 쿼리 실행 시 플러시가 자동 호출

**직접 호출**

-flush() 메소드를 직접 호출해서 강제로 플러시. 거의 테스트때만 사용

(로직상 먼저 commit 되어야 할 때도 필요할 것 같음)

**트랜잭션 커밋 시 플러시 자동 호출**

-트랜잭션을 커밋하기 전에 꼭 플러시를 호출해서 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영해야 한다. (JPA는 자동으로 호출)

**JPQL 쿼리 실행 시 플러시 자동 호출**

em.persist(member);  
List<Member> members = em.createQuery("select m from Member m",

-empersist는 영속성 컨텍스트에 있지만 아직 데이터베이스에 반영되지 않았다. 이때 JPQL을 실행하면 쿼리 결과가 조회되지 않기 때문에 (JPQL은 SQL로 변환되어 데이터베이스 조회)

자동으로 플러시를 호출 후, 조회 한다.

**3.5.1 플러시 모드 옵션**

-FlushModeType.AUTO : 커밋이나 쿼리를 실행할 때 플러시 (default)

-FlushModeType.COMMIT : 커밋할 때만 플러시

em.setFlushMode(FlushModeType.*COMMIT*);

**3.6. 준영속**

-준영속 상태의 엔티티는 영속성 컨텍스트가 제공하는 기능을 사용할 수 없다.

**준영속 상태로 만드는 3가지 방법**

1) em.detach : 특정 엔티티만 준영속 상태로 전환

2) em.clear() : 영속성 컨텍스트를 완전히 초기화 한다.

3) em.close() : 영속성 컨텍스트를 종료한다.

**3.6.1 엔티티를 준영속 상태로 전환 : detach()**

public void testDetached() {  
 EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("jpabook");  
 EntityManager em = emf.createEntityManager(); //엔티티 매니저 생성  
 EntityTransaction transaction = em.getTransaction();  
  
 Member member = new Member();  
 member.setId("memeberA");  
 member.setUsername("회원A");  
  
 // 회원 엔티티 영속 상태  
 em.persist(member);  
  
 // 회원 엔티티를 영속성 컨텍스트에서 분리, 준영속 상태  
 em.detach(member);  
 transaction.commit();  
}

-detach 호출 순간 1차 캐시부터 쓰기 지연 SQL 저장소까지 해당 엔티티를 관리하기 위한 모든 정보가 제거된다.

-쓰기 지연 SQL 저장소의 INSERT SQL도 제거되어서 데이터베이스에 저장되지도 않는다.

**3.6.2 영속성 컨텍스트 초기화 : clear()**

-em.detach()가 특정 엔티티 하나를 준영속으로 만들었다면 em.clear()는 모든 엔티티를 준영속 상태로 만든다.

Member member = em.find(Member.class, "memberA");  
em.clear();

member.setUsername(“changeName”);

-영속성 컨텍스트를 제거하고 새로 만든 것과 같다.

-당연히 set을 해도 데이터베이스에 반영되지 않는다.

**3.6.3 영속성 컨텍스트 종료 : close()**

-영속성 컨텍스트가 종료되어 관리되지 않는다. (준영속 상태)

**3.6.4 준영속 상태의 특징**

-거의 비영속 상태에 가깝다.

-식별자 값을 가지고 있다. (비영속과 차이)

-지연 로딩을 할 수 없다. (영속성 컨텍스트가 초기화)

**3.6.5 병합: merge()**

-준영속 상태의 엔티티를 다시 영속 상태로 변경하려면 병합을 사용한다. (새로운 영속 상태 반환)

-병합은 준영속, 비영속을 신경 쓰지 않는다. (save or update)

**CHAPTER4**

**JPA의 주요 어노테이션**

-객체와 테이블 매핑 : @Entity, @Table

-기본 키 매핑 : @Id

-필드와 컬럼 매핑 : @Column

-연관관계 매핑 : @manyToOne, @JoinColumn

**4.1 @Entity**

-테이블과 매핑할 클래스는 @Entity 어노테이션 필수

|  |  |  |
| --- | --- | --- |
| 속성 | 기능 | 기본값 |
| name | JPA에서 사용할 엔티티 이름 지정.  보통 기본값이 클래스 이름 | 클래스 이름 |

-기본 생성자 필수

🡺파라미터 생성자만 있는 경우 에러남

-final 클래스, enum, interface, inner 클래스에 사용 불가

-저장할 필드에 final 사용 금지

**4.2 @Table**

-엔티티와 매핑할 테이블을 지정한다. 생략 시 엔티티 이름을 테이블 이름으로 사용

|  |  |  |
| --- | --- | --- |
| 속성 | 기능 | 기본값 |
| name | 매핑할 테이블 이름 | 엔티티 이름을 사용 |
| catalog | Catalog 기능이 있는 데이터베이스에서 catalog를 매핑 |  |
| schema | Schema 기능이 있는 데이터베이스에서 schema를 매핑 |  |
| uniqueConstraints (DDL) | DDL 생성 시에 유니크 제약조건을 만든다.  (스키마 자동 생성 기능을 사용해서 DDL  만들때만 사용) |  |

**4.3 다양한 매핑 사용**

@Enumerated(EnumType.*STRING*)  
private RoleType type;  
  
@Temporal(TemporalType.*TIMESTAMP*)  
private Date createDate;  
  
@Temporal(TemporalType.*TIMESTAMP*)  
private Date lastModifiedDate;  
  
@Lob  
private String description;

-@Enumerated : 자바의 enum type 사용 시

-@Temporal : 자바의 날짜 타입을 사용 시

-@Lob : CLOB, BLOB 타입을 매핑할 수 있음

**4.4 데이터베이스 스키마 자동 생성**

persistence.xml

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

-애플리케이션 실행 시점에 데이터베이스 테이블을 자동으로 생성

<property name="hibernate.show\_sql" value="true" />

-해당 옵션 사용시 실행되는 DDL을 출력할 수 있음

**DDL 콘솔 출력**

Hibernate:  
 drop table MEMBER if exists  
Hibernate:  
 create table MEMBER (  
 ID varchar(255) not null,  
 age integer,  
 createDate timestamp,  
 description clob,  
 lastModifiedDate timestamp,  
 type varchar(255),  
 NAME varchar(255),  
 primary key (ID)  
 )

-기존 테이블을 삭제하고 다시 생성

-해당 DDL이 완벽하지 않으므로 개발 환경에서 사용하거나 매핑 참고용으로만 사용하는 것이 좋다.

**hibernate.hbm2ddl.auto 옵션**

|  |  |
| --- | --- |
| 옵션 | 설명 |
| create | 기존 테이블을 삭제하고 새로 생성 (DROP + CREATE) |
| create-drop | 애플리케이션을 종료할 때 생성한 DDL을 제거  (DROP + CREATE + DROP) |
| update | 데이터베이스 테이블과 엔티티 매핑정보를 비교해서 변경 사항만 수정 |
| validate | 데이터베이스 테이블과 엔티티 매핑정보를 비교해서 차이가 있으면 경고를 남기고 애플리케이션 미실행. |
| none | 자동 생성 기능을 사용하지 않는다. |

-create, create-drop, update 같은 옵션은 운영에서 사용하면 안 된다.

<property name="hibernate.ejb.naming\_strategy" value="org.hibernate.cfg.ImprovedNamingStrategy" />

-해당 전략을 사용하면 카멜 표기법을 테이블의 언더스코어 표기법으로 매핑

**결 과**

create table member (  
 id varchar(255) not null,  
 age integer,  
 create\_date timestamp,  
 description clob,  
 last\_modified\_date timestamp,  
 role\_type varchar(255),  
 name varchar(255) not null,  
 primary key (id)

**4.5 DDL 생성 기능**

@Column(name = "NAME", nullable = false, length = 10)  
private String username;

-@Column 매핑정보의 nullable 속성 값을 false로 하면 자동 생성되는 DDL에 not null 제약조건 추가 가능

-length 속성 값을 사용하면 자동 생성되는 DDL에 문자의 크기를 지정

**결 과**

create table member (  
 id varchar(255) not null,  
 age integer,  
 create\_date timestamp,  
 description clob,  
 last\_modified\_date timestamp,  
 role\_type varchar(255),  
 name varchar(10) not null,  
 primary key (id)

@Entity  
@DynamicUpdate  
@Table(name = "MEMBER", uniqueConstraints = { @UniqueConstraint(  
 name = "NAME\_AGE\_UNIQUE",  
 columnNames = { "NAME", "AGE" }) })  
public class Member {

-uniqueConstraints 추가

**결 과**

Hibernate:  
 alter table member  
 add constraint NAME\_AGE\_UNIQUE unique (name, age)

-@Column의 length와 nullable 속성들은 DDL을 자동 생성할 때만 사용되고 JPA의 실행 로직에는 영향을 주지 않는다.

🡺하지만 개발자가 엔티티만 보고도 손쉽게 다양한 제약 조건을 파악할 수 있다는 이점은 있다.

**4.6 기본 키 매핑**

JPA가 제공하는 데이터베이스 기본 키 생성 전략

-직접 할당 : 기본 키를 애플리케이션에서 직접 할당

-자동 생성 : 대리 키 사용 방식 (@GeneratedValue)

* IDENTITY : 기본 키 생성을 데이터베이스에 위임한다. (Mysql AutoIncrement)
* SEQUENCE : 데이터베이스 시퀀스를 사용해서 기본 키를 할당한다. (Oracle Sequence)
* TABLE : 키 생성 테이블을 사용한다.

🡺여기서 하나 궁금한게 mysql을 사용하다가 oracle로 변경될 경우 자동 생성 방식을 변경해 주어야 하는가? (데이터베이스에 의존적)

<property name="hibernate.id.new\_generator\_mappings" value="true" />

-GeneratedValue를 사용하려면 해당 옵션을 추가해 주어야 한다.

**4.6.1 기본 키 직접 할당 전략**

@Id  
@Column(name = "ID")  
private String id;

**ID 적용 가능 자바 타입**

-자바 기본형

-자바 래퍼형

-String

-java.util.Date

-java.sql.Date

-java.math.BigDecimal

-java.math.BigInteger

Member member = new Member();  
member.setId(“id1”);

🡺직접 할당해 주어야 한다.

**4.6.2 IDENTITY 전략**

-MySQL, PostgreSQL, SQL Server, DB2에서 사용한다.

CREATE TABLE BOARD (  
 ID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,  
 DATA VARCHAR(255)  
);

-auto\_increment 기능을 수행

@Id  
@GeneratedValue(strategy = GenerationType.*IDENTITY*)  
private Long id;

🡺기본 키 생성을 데이터베이스에 위임하는 전략이다.

**주의**

-IDENTITY 식별자 생성 전략은 엔티티를 데이터베이스에 저장해야 식별자(auto\_increment)를 구할 수 있으므로 em.persist()를 호출하는 즉시 INSERT SQL이 데이터베이스 전달된다.

-이 전략은 트랜잭션을 지원하는 쓰기 지연이 동작하지 않는다.

**4.6.3 SEQUENCE 전략**

-오라클 PostgreSQL, DB2, H2에서 사용한다.

CREATE TABLE BOARD (  
 ID BIGINT NOT NULL PRIMARY KEY,  
 DATA VARCHAR(255)  
);  
  
CREATE SEQUENCE BOARD\_SEQ START WITH 1 INCREMENT BY 1;

-시퀀스 생성 (START WITH로 시작 숫자를 설정할 수 있다.)

@Entity  
@SequenceGenerator(  
 name = "BOARD\_SEQ\_GENERATOR",  
 sequenceName = "BOARD\_SEQ", // 매핑할 데이터베이스 시퀀스 이름  
 initialValue = 1,  
 allocationSize = 1  
)  
public class Board {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*SEQUENCE*,

generator = "BOARD\_SEQ\_GENERATOR")  
 private Long id;  
}

-데이터베이스 시퀀스는 유일한 값을 순서대로 생성하는 특별한 데이터베이스 오브젝트이다.

-사용할 데이터베이스 시퀀스를 매핑해야 한다.

-SequenceGenerator를 사용해서 시퀀스를 생성했다. sequenceName은 실제 데이터베이스 시퀀스와 매칭시킨다.

-키 전략을 GenerationType.SEQUENCE로 설정하고 generator를 선택한다.

-IDENTITY는 저장이 일어나면 트랜잭션 상관 없이 바로 데이터베이스에 저장했다. (지연 동작X)

반면에 SEQUENCE 전략은 저장이 일어나면 데이터베이스 시퀀스를 사용해서 식별자를 조회 후,식별자를 엔티티에 할당한 다음 영속성 컨텍스트에 저장.

이후 트랜잭션을 커밋해서 플러시가 일어나면 데이터베이스에 저장한다.

**@SequenceGenerator**

|  |  |  |
| --- | --- | --- |
| 속성 | 기능 | 기본값 |
| name | 식별자 생성기 이름 | 필수 |
| sequenceName | 데이터베이스에 등록되어 있는 시퀀스 이름 | Hibernate\_sequence |
| initialValue | DDL 생성시에만 사용됨, 처음 시작하는 수를 시정 | 1 |
| allocationSize | 시퀀스 한 번 호출에 증가하는 수 | 50 |
| catalog, schema | 데이터베이스 catalog, schema 이름 |  |

ex) create sequence [**sequenceName**] start with [**initialValue**] increment by [**allocationSize**]

**주의**

-allocationSize가 50인 이유는 최적화 때문이다. allocationSize 값이 50이면 시퀀스를 한번에 50 증가 시킨다음 1~50까지는 메모리에서 식별자를 할당한다.

-단점으로는 아래와 같이 시퀀스가 한번에 많이 증가한다.



**4.6.4 TABLE전략**

-TABLE 전략은 키 생성 전용 테이블을 하나 만들고 여기에 이름과 값으로 사용할 컬럼을 만들어 데이터베이스 시퀀스를 흉내 내는 전략이다. (모든 데이터베이스에 적용 가능)

**TABLE 생성**

CREATE TABLE MY\_SEQUENCES (  
 sequence\_name varchar(255) not null,  
 next\_val bigint,  
 primary key (sequence\_name)  
 )

🡺TableGenerator의 pkColumnName 기본값이 **sequence\_name**, valueColumnName이 **next\_val**

**TABLE 전략 매핑코드**

@Entity  
@TableGenerator(  
 name = "BOARD\_SEQ\_GENERATOR",  
 table = "MY\_SEQUENCES",  
 pkColumnValue = "BOARD\_SEQ",  
 allocationSize = 1  
)  
public class Board {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*TABLE*, generator = "BOARD\_SEQ\_GENERATOR")  
 private Long id;

**@TableGenerator**

|  |  |  |
| --- | --- | --- |
| 속성 | 기능 | 기본값 |
| name | 식별자 생성기 이름 | 필수 |
| table | 키생성 테이블명 | hibernate\_sequences |
| valueColumnName | 시퀀스 값 컬럼명 | Next\_val |
| pkColumnName | 시퀀스 컬럼명 | sequence\_name |
| initialValue | 초기 값, 마지막으로 생성된 값이 기준 | 0 |
| allocationSize | 시퀀스 한 번 호출에 증가하는 수 | 50 |
| catalog, schema | 데이터베이스 catalog, schema 이름 |  |
| uniqueConstraints(DDL) | 유니크 제약 조건을 지정할 수 있다 |  |

**참고**

SEQUENCE 전략과 비교해서 데이터베이스와 한 번 더 통신한다.

**4.6.5 AUTO 전략**

-Auto는 데이터베이스 방언에 따라서 IDENTITY, SEQUENCE, TABLE 전략 중 하나를 자동으로 선택한다.

-Auto 장점은 데이터베이스를 변경해도 코드를 수정할 필요가 없다.

-SEQUENCE나 TABLE 전략이 선택되면 시퀀스나 키 생성용 테이블을 미리 만들어 두어야 한다.

자동 생성 기능 사용 시, 하이버네이트가 만들어줌

**4.6.6 기본 키 매핑 정리**

-직접 할당 : 저장 하기 전에 애플리케이션에서 직접 식별자 값을 할당.

-SEQUENCE : 데이터베이스 시퀀스에서 식별자 값을 획득 후 영속성 컨텍스트에 저장

-TABLE : 시퀀스 생성용 테이블에서 식별자 값을 획득한 후 영속성 컨텍스트에 저장

-IDENTITY : 엔티티를 저장할 때 식별자 값이 생성되는데 획득 후 영속성 컨텍스트에 저장

**참고**

-자연키 : 비즈니스에 의미가 있는 키 (주민등록번호, 이메일)

-대리키 : 비즈니스와 관련 없는 임의키 (오라클 시퀀스 auto\_increment, 키생성 테이블)

🡺자연키보다 대리키를 사용하자. 자연키는 비즈니스 환경에 따라서 변경될 수 있다. (수정범위 커짐)

**4.7 필드와 컬럼 매핑: 레퍼런스**

책 참고 (P.145)

**CHAPTER5**

-객체 관계 매핑(orm)에서 가장 어려운 부분이 바로 객체 연관관계와 테이블 연관관계를 매핑하는 일이다.

**핵심 키워드**

-방향(Direction) : 단방향, 양방향이 있다.

-다중성(Multiplicity) : 다대일(N:1), 일대다(1:N), 일대일(1:1), 다대다(N:M) 다중성이 있다.

-연관관계의 주인(owner) : 객체를 양방향 연관관계로 만들면 연관관계의 주인을 정해야 한다.

**5.1 단방향 연관관계**

-객체 연관관계

* 회원 객체와 팀 객체는 **단방향 관계** (member -> team 조회 가능, team -> member 조회 불가능)

-테이블 연관관계

* 회원 테이블과 팀 테이블은 **양방향 관계** (member 테이블의 team\_id 외래키 하나로 member join team과 team join member 둘다 가능)

-객체에서 양방향 관계는 엄밀히 말하면 서로 다른 단방향 관계 2개다.

-객체 연관관계 vs 테이블 연관관계

* 객체는 참조로 연관관계를 맺는다.
* 테이블은 외래 키로 연관관계를 맺는다.

**5.1.1 순수한 객체 연관관계**

-객체 그래프 탐색 : 객체 참조를 사용해서 연관관계를 탐색할 수 있다.

**5.1.2 테이블 연관관계**

**Member**

@Entity  
public class Member {  
 @Id  
 @Column(name = "MEMBER\_ID")  
 private String id;  
 private String username;  
  
 //연관관계 매핑  
 @ManyToOne  
 @JoinColumn(name = "TEAM\_ID")  
 private Team team;

…

}

**Team**

@Entity  
public class Team {  
 @Id  
 @Column(name = "TEAM\_ID")  
 private String id;  
 private String name;

-객체 연관관계 : 회원 객체의 member.team 필드 사용

-테이블 연관관계 : 회어ㅜㄴ 테이블의 MEMBER.TEAM\_ID 외래 키 컬럼 사용

-@ManyToOne : 이름 그대로 다대일(N:1) 관계라는 매핑 정보. 연관관계 매핑 시 필수.

-@JoinColumn(name=”TEAM\_ID”) : 조인 컬럼은 외래 키를 매핑할 때 사용한다.

**5.1.4 @JoinColumn**

-외래키를 매핑할 때 사용한다.

|  |  |  |
| --- | --- | --- |
| 속성 | 기능 | 기본값 |
| name | 매핑할 외래 키 이름 | 필드명 +\_+ 참조하는 테이블의 기본키 컬럼명 |
| referencedColumnName | 외래 키가 참조하는 대상 테이블의 컬럼명 | 참조하는 테이블의 기본키 컬럼명 |
| foreignKey(DDL) | 외래 키 제약조건을 직접 지정 |  |

-JoinColumn 생략 시, 기본 전략사용 (필드명 +\_+ 참조하는 테이블의 컬럼명)

**5.1.5 @ManyToOne**

-다대일 관계에서 사용한다.

|  |  |  |
| --- | --- | --- |
| 속성 | 기능 | 기본값 |
| optional | false로 설정하면 연관된 엔티티가 항상 있어야 한다. | true |
| fetch | 글로벌 페치 전략 설정 | [-@ManyToOne=FetchType.EAGER](mailto:-@ManyToOne=FetchType.EAGER)  [-@ManyToOne=FetchType.LAZY](mailto:-@ManyToOne=FetchType.LAZY) |
| cascade | 영속성 전이 기능을 사용한다. |  |
| targetEntity | 연관된 엔티티의 타입 정보를 설정 |  |

**5.2. 연관관계 사용**

**5.2.1 저장**

**MEMBER**



**TEAM**



**5.2.2 조회**

-연관관계 엔티티 조회 방법

* 객체 그래프 탐색
* 객체지향 쿼리 사용 (JPQL)

-객체 그래프 탐색

Member member = em.find(Member.class, "member1");  
Team team = member.getTeam(); // 객체 그래프 탐색  
System.*out*.println("팀 이름 = " + team.getName());

-객체지향 쿼리 사용

String jpql = "select m from Member m join m.team t where " +  
 "t.name=:teamName";  
  
List<Member> resultList = em.createQuery(jpql, Member.class)  
 .setParameter("teamName", "팀1")  
 .getResultList();  
  
for (Member member : resultList) {  
 System.*out*.println("[query] member.username=" + member.getUsername());  
}

**5.2.3 수정**

// 새로운 팀2  
Team team2 = new Team("team2", "팀2");  
em.persist(team2);  
  
// 회원 1에 새로운 팀2 설정  
Member member = em.find(Member.class, "member1");  
member.setTeam(team2);

-엔티티의 값만 변경해두면 트랜잭션을 커밋할 때 플러시가 일어나면서 변경 감지 기능이 작동한다.

**5.2.4 연관관계 제거**

private void deleteRelation(EntityManager em) {  
 Member member1 = em.find(Member.class, "member1");  
 member1.setTeam(null); // 연관관계 제거  
}

-연관관계를 null로 설정

**5.2.5. 연관된 엔티티 삭제**

//삭제시 연관관계 제거해 주어야 함.  
member3.setTeam(null);  
em.remove(team3);

**5.3 양방향 연관관계**

-JPA는 List를 포함해서 Collection, Set. Map 같은 다양한 컬렉션을 지원한다.

// 양방향 엔티티  
@OneToMany(mappedBy = "team")  
private List<Member> members = new ArrayList<Member>();

-mappedBy 속성은 양방향 매핑일 때 사용하는데 반대쪽 매핑의 필드 이름을 값으로 주면 된다.

**5.4. 연관관계의 주인**

-객체 연관관계

* 회원 -> 팀 연관관계 1개(단방향)
* 팀 -> 회원 연관관계 1개 (단방향)

-테이블

* 회원 <-> 팀의 연관관계 1개(양방향)

-엔티티를 양방향 연관관계로 설정하면 객체의 참조는 둘인데 외래 키는 하나다. 그렇다면 둘 중 어떤 관계를 사용해서 외래키를 관리해야 할까?

🡺이런 차이 때문에 두 객체 연관관계 중 하나를 정해서 테이블의 외래키를 관리해야 하는데 이것을 연관관계 주인이라고 한다. (mappedBy가 필요한 이유)

**5.4.1 양방향 매핑의 규칙: 연관관계의 주인**

-연관관계의 주인만이 데이터베이스 연관관계와 매핑되고 외래 키를 관리(등록, 수정, 삭제) 할 수 있다. 반면에 주인이 아닌 쪽은 읽기만 할 수 있다.

-주인은 mappedBy 속성을 사용하지 않는다.

-주인이 아니면 mappedBy 속성을 사용해서 속성의 값으로 연관관계의 주인을 지정해야 한다.

**5.4.2 연관관계의 주인은 외래 키가 있는 곳**

**참고**

데이터베이스 테이블의 다대일, 일대다 관계에서는 항상 다 쪽이 외래 키를 가진다.

다 쪽인 @ManyToOne은 항상 연관관계의 주인이 되므로 mappedBy를 설정할 수 없다.

**5.5 양방향 연관관계 저장**

Member member3 = new Member("member3", "회원3");  
team.getMembers().add(member3);

🡺주인이 아닌 곳에서 입력된 값이므로 무시된다.

**5.6 양방향 연관관계의 주의점**

private void testSaveNonOwner(EntityManager em) {  
 // 회원 3 저장  
 Member member3 = new Member("member3", "회원3");  
 em.persist(member3);  
  
 Member member4 = new Member("member4", "회원4");  
 em.persist(member4);  
  
 Team team3 = new Team("team3", "팀3");  
 // 주인이 아닌 곳만 연관관계 설정  
 team3.getMembers().add(member3);  
 team3.getMembers().add(member4);  
  
 em.persist(team3);  
}

🡺주인이 아닌 곳만 연관관계 설정



-연관관계의 주인만이 외래 키의 값을 변경할 수 있다.

**5.6.1 순수한 객체까지 고려한 양방향 연관관계**

-객체 관점에서 양쪽 방향에(주인, 주인 아닌곳) 모두 값을 입력해주는 것이 가장 안전하다.

private void testORM\_양방향(EntityManager em) {  
 Team team5 = new Team("team5", "팀5");  
 em.persist(team5);  
  
 Member member6 = new Member("member6", "회원6");  
  
 // 양방향 연관관계 설정  
 member6.setTeam(team5); //연관관계 설정 member -> team  
 team5.getMembers().add(member6); //연관관계 설정 team -> member  
 em.persist(member6);  
  
 Member member7 = new Member("member7", "회원7");  
  
 // 양방향 연관관계 설정  
 member7.setTeam(team5); //연관관계 설정 member -> team  
 team5.getMembers().add(member7); //연관관계 설정 team -> member  
 em.persist(member7);  
}

**5.6.2 연관관계 편의 메소드**

public void setTeam(Team team) {  
 this.team = team;  
 team.getMembers().add(this);  
}

🡺실수를 줄이기 위해 Member 클래스 setTeam에 양방향으로 설정

**5.6.3 연관관계 편의 메소드 작성 시 주의사항**

public void setTeam(Team team) {  
 // 기존 팀과 관계를 제거  
 if (this.team != null) {  
 this.team.getMembers().remove(this);  
 }  
  
 this.team = team;  
 team.getMembers().add(this);  
}

🡺여러 번 추가 할 때 기존 연관관계가 남아있기 때문에 삭제해 주어야 한다.

**5.7 정리**

-단방향 매핑만으로 테이블과 객체의 연관관계 매핑은 이미 완료되었다.

-단방향을 양방향으로 만들면 반대방향으로 객체 그래프 탐색 기능이 추가된다.

-양방향 연관관계를 매핑하려면 객체에서 양쪽 방향을 모두 관리해야 한다.

**CHAPTER6**

-엔티티의 연관관계 매핑 시 다음 3가지 고려

* 다중성 (일대일, 일대다, 다대일, 다대다)
* 단방향, 양방향
* 연관관계의 주인 (양방향 관계시만)

**6.1 다대일**

-일,다 관계에서 외래 키는 항상 다쪽에 있다. 연관관계의 주인은 항상 다쪽이다.

**6.1.1 다대일 단방향[N:1]**

@Entity  
public class Member {  
 @Id @GeneratedValue  
 @Column(name = "MEMBER\_ID")  
 private Long id;  
  
 private String username;  
  
 @ManyToOne  
 @JoinColumn(name = "TEAM\_ID")  
 private Team team;

@Entity  
public class Team {  
 @Id @GeneratedValue  
 @Column(name = "TEAM\_ID")  
 private Long id;  
  
 private String name;

-회원은 팀 엔티티를 참조할 수 있지만 반대로 팀에서 회원을 참조하는 필드가 없다.

(다대일 단방향 연관관계)

**6.1.2 다대일 양방향 [N:1. 1:N]**

@Entity  
public class Member {  
 @Id @GeneratedValue  
 @Column(name = "MEMBER\_ID")  
 private Long id;  
  
 private String username;  
  
 @ManyToOne  
 @JoinColumn(name = "TEAM\_ID")  
 private Team team;

@Entity  
public class Team {  
 @Id @GeneratedValue  
 @Column(name = "TEAM\_ID")  
 private Long id;  
  
 private String name;  
  
 @OneToMany(mappedBy = "team")  
 private List<Member> members = new ArrayList<Member>();  
  
 public void addMember(Member member) {  
 this.members.add(member);  
 if (member.getTeam() != this) {  
 member.setTeam(this);  
 }  
 }

-양방향은 외래 키가 있는 쪽이 연관관계의 주인이다.

-양방향 연관관계는 항상 서로 참조해야 한다.

**6.2 일대다**

-다대일 관계의 반대 방향이다.

**6.2.1 일대다 단방향[1:N]**

@Entity  
public class Team {  
 @Id @GeneratedValue  
 @Column(name = "TEAM\_ID")  
 private Long id;  
  
 private String name;  
  
 @OneToMany  
 @JoinColumn(name = "TEAM\_ID") // Member 테이블의 Team\_id (fk)  
 private List<Member> members = new ArrayList<Member>();

@Entity  
public class Member {  
 @Id @GeneratedValue  
 @Column(name = "MEMBER\_ID")  
 private Long id;  
  
 private String username;

-일대 단방향 관계는 약간 특이한데 보통 자신이 매핑한 테이블의 외래 키를 관리하는데, 이 매핑은 반대쪽 테이블에 있는 외래 키를 관리한다.

🡺일대다 관계에서 외래 키는 항상 다쪽 테이블에 있다. 하지만 다쪽인 Member 엔티티에는 외래 키를 매핑하 수 있는 참조필드가 없다.

-일대다 단방향 매핑의 단점

private void testSave(EntityManager em) {  
 Member member1 = new Member("member1");  
 Member member2 = new Member("member2");  
  
 Team team1 = new Team("team1");  
 team1.getMembers().add(member1);  
 team1.getMembers().add(member2);  
  
 em.persist(member1); // Insert -member1  
 em.persist(member2); // Insert -member2  
 em.persist(team1); // Insert -team1, update -member1, member2  
}

🡺본인 테이블에 외래 키가 있으면 엔티티의 저장과 연관관계 처리를 INSERT SQL 한번으로 끝낼 수 있지만, 다른 테이블에 외래 키가 있으면 연관관계 처리를 위한 UPDATE SQL을 추가로 실행해야 한다.

-Member 엔티티는 Team 엔티티를 모르고 연관관계도 Team에서 관리한다. 따라서 Member 엔티티를 저장할 때는 TEAM\_ID에 아무것도 저장되지 않는다. 대신 Team을 저장할 때 Team.members의 참조 값을 확인해서 Member에 TEAM\_ID를 업데이트 한다.

-일대다 단방향 보다 다대일 양방향 매핑을 사용하자.

**6.2.2 일대다 양방향 [1:N, N:1]**

-일대다 양방향 매핑은 존재하지 않는다. 대신 다대일 양방향 매핑을 사용해야 한다.

@Entity  
public class Team {  
 @Id @GeneratedValue  
 @Column(name = "TEAM\_ID")  
 private Long id;  
  
 private String name;  
  
 @OneToMany  
 @JoinColumn(name = "TEAM\_ID")  
 private List<Member> members = new ArrayList<Member>();

@Entity  
public class Member {  
 @Id @GeneratedValue  
 @Column(name = "MEMBER\_ID")  
 private Long id;  
  
 private String username;  
  
 @ManyToOne  
 @JoinColumn(name = "TEAM\_ID", insertable = false, updatable = false)  
 private Team team;

🡺둘 다 같은 키를 관리하므로 문제가 발생할 수 있다. 따라서 반대편인 다대일 쪽은

Insertable = false, updatable = false로 설정해서 읽기만 가능하도록 . .

-되도록 다대일 양방향 매핑을 사용하자.

**6.3 일대일 [1:1]**

-일대일 관계는 그 반대도 일대일 관계다.

-테이블 관계에서 일대다, 다대일은 항상 다(N)쪽이 외래 키를 가진다.

반면 일대일 관계는 주 테이블이나 대상 테이블 둘 중 어느 곳이나 외래 키를 가질 수 있다.

**주 테이블에 외래키**

-외래 키를 객체 참조와 비슷하게 사용할 수 있어서 객체지향 개발자들이 선호. 장점은 주 테이블만 확인해도 대상 테이블과 연관관계가 있는지 알 수 있다.

**대상 테이블에 외래키**

-전통적인 데이터베이스 개발자들이 선호. 장점은 일대일에서 일대다로 변경할 때 테이블 구조를 그대로 유지할 수 있다.

**6.3.1 주 테이블에 외래 키**

**단방향**

@Entity  
public class Member {  
 @Id @GeneratedValue  
 @Column(name = "MEMBER\_ID")  
 private Long id;  
  
 private String username;  
  
 @OneToOne  
 @JoinColumn(name = "LOCKER\_ID")  
 private Locker locker;

@Entity  
public class Locker {  
 @Id @GeneratedValue  
 @Column(name = "LOCKER\_ID")  
 private Long id;  
 private String name;

-이 관계는 다대일 단방향(@ManyToOne)과 거의 비슷

**양방향**

@Entity  
public class Member {  
 @Id @GeneratedValue  
 @Column(name = "MEMBER\_ID")  
 private Long id;  
  
 private String username;  
  
 @OneToOne  
 @JoinColumn(name = "LOCKER\_ID")  
 private Locker locker;

@Entity  
public class Locker {  
 @Id @GeneratedValue  
 @Column(name = "LOCKER\_ID")  
 private Long id;  
 private String name;  
  
 @OneToOne(mappedBy = "locker")  
 private Member member;

-양방향이기 때문에 연관관계의 주인을 정해야 한다. Member가 외래 키를 가지고 있으므로 연관관계의 주인이다.

**6.3.2 대상 테이블에 외래 키**

**단방향**

-JPA에서 지원 안 한다.

**양방향**

@Entity  
public class Member {  
 @Id @GeneratedValue  
 @Column(name = "MEMBER\_ID")  
 private Long id;  
  
 private String username;  
  
 @OneToOne(mappedBy = "member")  
 private Locker locker;

@Entity  
public class Locker {  
 @Id @GeneratedValue  
 @Column(name = "LOCKER\_ID")  
 private Long id;  
 private String name;  
  
 @OneToOne  
 @JoinColumn(name = "MEMBER\_ID")  
 private Member member;

-대상 테이블에 외래 키를 두고 싶은 경우

**6.4 다대다 [N:N]**

-관계형 데이터베이스는 정규화된 테이블 2개로 다대다 관계를 표현할 수 없다. (중간에 연결 테이블 필요)

@Entity  
public class Member {  
 @Id @GeneratedValue  
 @Column(name = "MEMBER\_ID")  
 private String id;  
  
 private String username;  
  
 @ManyToMany  
 @JoinTable(name = "MEMBER\_PRODUCT",  
 joinColumns = @JoinColumn(name = "MEMBER\_ID"),  
 inverseJoinColumns = @JoinColumn(name = "PRODUCT\_ID"))  
 private List<Product> products = new ArrayList<Product>();

@Entity  
public class Product {  
 @Id @Column(name = "PRODUCT\_ID")  
 private String id;  
 private String name;

-JoinTable을 사용하면 회원\_상품(Member\_Product) 엔티티 없이 매핑을 완료할 수 있다.

-@JoinTable.name : 연결 테이블 지정

[-@JoinTable.joinColumns](mailto:-@JoinTable.joinColumns) : 현재 방향인 회원과 매핑할 조인 컬럼 정보를 지정

[-@JoinTable.inverseJoinColumns](mailto:-@JoinTable.inverseJoinColumns) : 반대 방향인 상품과 매핑할 조인 컬럼 정보를 지정

-@ManyToMany로 매핑하면 자동으로 연결테이블을 생성해 준다.

private void find(EntityManager em) {  
 Member member = em.find(Member.class, "member1");  
 List<Product> products = member.getProducts(); //객체 그래프 탐색  
 for (Product product : products) {  
 System.*out*.println("product.name = " + product.getName());  
 }  
}

-find를 하면 다음과 같이 member\_product 테이블을 이용한다.

|  |
| --- |
| SELECT \* FROM MEMBER\_PRODUCT MP  INNER JOIN PRODUCT P ON MP.PRODUCT\_ID=P.PRODUCT\_ID  WHERE MP.MEMBER\_ID=? |

**6.4.2 다대다: 양방향**

-역방향도 @ManyToMany를 사용한다.

@Entity  
public class Product {  
 @Id  
 private String id;  
 private String name;  
  
 @ManyToMany(mappedBy = "products") // 역방향 추가  
 private List<Member> members = new ArrayList<Member>();

**6.4.3 다대다: 매핑의 한계와 극복, 연결 엔티티 사용**

-@ManyToMany를 사용하면 연결 테이블을 자동 처리해주지만 다른 컬럼을 추가할 수 없다.

**복합 기본 키**

@Entity  
@IdClass(MemberProductId.class)  
public class MemberProduct {

...

}

public class MemberProductId implements Serializable {  
 private String member; //MemberProduct.member와 연결  
 private String product; //MemberProduct.product와 연결

}

-기본 키가 MEMBER\_ID와 PRODUCT\_ID로 이루어진 복합 기본키다.

-@IdClass를 사용해서 복합 키를 사용할 식별자 클래스를 지정

**식별자 클래스 특징**

-복합 키는 별도의 식별자 클래스로 만들어야 된다.

-Serializable을 구현

-euqals와 hashCode 메소드를 구현해야 한다.

-기본 생성자가 있어야 한다.

-식별자 클래스는 public 이어야 한다.

**식별 관계**

-부모 테이블의 기본 키를 받아서 자신의 기본 키 + 외래 키로 사용하는 것을 데이터 용어로 식별 관계라 한다.

-복합 키는 항상 식별자 클래스를 만들어야 한다.

**6.4.4 다대다: 새로운 기본 키 사용**

-데이터베이스에서 자동으로 생성해주는 대리 키를 Long 값으로 사용하면 영구적이고 비즈니스에 의존하지 않는다. 그리고 복합 키를 만들지 않아도 된다.

@Entity  
public class Order {  
 @Id @GeneratedValue // 대리키  
 @Column(name = "ORDER\_ID")  
 private Long id;  
  
 @ManyToOne  
 @JoinColumn(name = "MEMBER\_ID")  
 private Member member;  
  
 @ManyToOne  
 @JoinColumn(name = "PRODUCT\_ID")  
 private Product product;

**6.4.5 다대다 연관관계 정리**

-식별 관계 : 받아온 식별자를 기본 키 + 외래 키로 사용

-비식별 관계 : 받아온 식별자는 외래 키로만 사용하고 새로운 식별자를 추가

**CHAPTER7**

-상속 관계 매핑 : 객체의 상속 관계를 데이터베이스에 어떻게 매핑하는지

-@MappedSuperclass : 등록일, 수정일 같이 공통으로 사용하는 매핑 정보만 상속 받고 싶을 때

-복합 키와 식별 관계 매핑 : 데이터베이스의 식별자가 하나 이상일 때 매핑하는 방법

-조인 테이블 : 연관관계를 관리하는 연결 테이블을 사용할 때 이 테이블을 매핑하는 방법

-엔티티 하나에 여러 테이블 매핑하기 : 엔티티 하나에 여러 테이블을 매핑

**7.1 상속 관계 매핑**

-데이터베이스의 슈퍼타입 서브타입 관계를 매핑한다. (상속 개념과 유사)

* 각각의 테이블로 변환 : 각각을 모두 테이블로 만들고 조회 할 때 조인 (JPA는 조인 전략)
* 통합 테이블로 변환 : 테이블을 하나만 사용해서 통합한다. (JPA는 단일 테이블 전략)
* 서브타입 테이블로 변환 : 서브 타입마다 하나의 테이블을 만든다. (JPA는 구현 클래스마다 테이블 전략)

**7.1.1 조인 전략**

-엔티티 각각을 모두 테이블로 만들고 자식 테이블이 부모 테이블의 기본키를 받아서 기본키 + 외래키로 사용하는 전략

-주의점은 객체는 타입으로 구분할 수 있지만 테이블은 없기 때문에 타입을 구분하는 컬럼을 추가해야 한다.



**부모**

@Entity  
@Inheritance(strategy = InheritanceType.*JOINED*)  
@DiscriminatorColumn(name = "DTYPE")  
public abstract class Item {  
 @Id @GeneratedValue  
 @Column(name = "ITEM\_ID")  
 private Long id;  
  
 private String name; // 이름  
 private int price; // 가격  
}

**자식**

@Entity  
@DiscriminatorValue("A")  
public class Album extends Item {  
 private String artist;  
}

**-장점**

* 테이블이 정규화된다.
* 외래 키 참조 무결성 제약조건을 활용할 수 있다.
* 저장공간의 효율적으로 사용한다.

**-단점**

* 조회할 때 조인이 많이 사용되므로 성능이 저하될 수 있다.
* 조회 쿼리가 복잡하다.
* 데이터를 등록할 INSERT SQL을 두 번 실행한다.

**-특징**

* JPA 표준 명세는 구분 컬럼을 사용하도록 하지만, 몇몇 구현체는 구분 컬럼(@DiscriminatorColumn) 없이도 동작한다.

**7.1.2 단일 테이블 전략**

-테이블을 하나만 사용하고 구분 컬럼으로 어떤 자식 데이터가 저장되었는지 구분한다.

-조회할 때 조인을 사용하지 않으므로 일반적으로 가장 빠르다.

-주의점은 자식 엔티티가 매핑한 컬럼은 모두 null을 허용해야 한다. (자식별로 사용 컬럼이 다르기 때문)



**부모**

@Entity  
@Inheritance(strategy = InheritanceType.*SINGLE\_TABLE*)  
@DiscriminatorColumn(name = "DTYPE")  
public abstract class Item {  
 @Id @GeneratedValue  
 @Column(name = "ITEM\_ID")  
 private Long id;  
 private String name; // 이름  
 private int price; // 가격

}

**자식**

@Entity  
@DiscriminatorValue("A")  
public class Album extends Item {  
 private String artist;  
}

**-장점**

* 조인이 필요 없으므로 일반적으로 조회 성능이 빠르다.
* 조회 쿼리가 단순하다.

**-단점**

* 자식 엔티티가 매핑한 컬럼은 모두 null을 허용해야 한다. (데이터베이스에서)
* 단일 테이블에 모든 것을 저장하기 때문에 테이블이 커질 수 있다. 그러므로 성능이 오히려 느려질 수 있다.

**-특징**

* 구분 컬럼을 꼭 사용해야 한다. (@DiscriminatorColumn) 꼭 설정
* @DiscriminatorValue를 지정하지 않으면 기본으로 엔티티 이름을 사용

**7.1.3 구현 클래스마다 테이블 전략**

-자식 엔티티마다 테이블을 만든다. 그리고 자식 테이블 각각에 필요한 컬럼이 모두 있다.

-자식 엔티티마다 테이블을 만들기 때문에 일반적으로 추천하지 않는다.



**부모**

@Entity  
@Inheritance(strategy = InheritanceType.*TABLE\_PER\_CLASS*)  
public abstract class Item {  
 @Id  
 @GeneratedValue  
 @Column(name = "ITEM\_ID")  
 private Long id;  
 private String name; // 이름  
 private int price; // 가격

}

**자식**

@Entity  
@DiscriminatorValue("A")  
public class Album extends Item {  
 private String artist;  
}

**-장점**

* 서브 타입을 구분해서 처리할 때 효과적이다.
* not null 제약조건을 사용할 수 있다.

**-단점**

* 여러 자식 테이블을 함께 조회할 때 성능이 느리다(SQL 에 UNION을 사용)
* 자식 테이블을 통합해서 쿼리하기 어렵다.

**-특징**

* 구분 컬럼을 사용하지 않는다.

**7.2 @MappedSuperclass**

-부모 클래스는 테이블과 매핑하지 않고 부모 클래스를 상속받는 자식 클래스에게 매핑 정보만 제공하고 싶으면 @MappedSuperclass를 사용하면 된다.

-@MappedSuperclass는 추상 클래스와 비슷하다.

@MappedSuperclass  
public abstract class BaseEntity {  
 @Id  
 @GeneratedValue  
 private Long id;  
 private String name;

}

@Entity  
public class Member extends BaseEntity {  
 //ID 상속  
 //NAME 상속  
 private String email;

}

@Entity  
public class Seller extends BaseEntity {  
 //ID 상속  
 //NAME 상속  
 private String shopName;

}

-BaseEntity는 테이블과 매핑할 필요가 없고 자식 엔티티에게 공통으로 사용되는 매핑 정보만 제공하면 된다.

-부모로부터 물려받은 매핑 정보를 재정의 하려면 @AtrributeOverrides나 @AttributeOverride를 사용하고 연관관계를 재정의하려면 @AssociationOverrides나 @AssociationOverride를 사용한다.

**@MappedSuperclass의 특징**

-테이블과 매핑되지 않고 자식 클래스에 엔티티의 매핑 정보를 상속하기 위해 사용

-@MappedSuperclass로 지정한 클래스는 엔티티가 아니므로 em.find()나 JPQL에서 사용할 수 없다.

-이 클래스를 직접 생성해서 사용할 일은 거의 없으므로 추상 클래스로 만드는 것을 권장

**7.3. 복합 키와 식별 관계 매핑**

**7.3.1 식별 관계 vs 비식별 관계**

**식별 관계**

-부모 테이블의 기본 키를 내려받아서 잣기 테이블의 기본 키 + 외래키로 사용하는 관계다.

(기본 키(PK) + 외래 키(FK))

**비식별 관계**

-부모 테이블의 기본 키를 받아서 자식 테이블의 외래 키로만 사용하는 관계다. (외래 키(FK)

-비식별 관계는 외래 키에 NULL을 허용하는지에 따라 필수적 빗기별 관계와 선택적 비식별 관계로 나눈다.

* 필수적 비식별 관계(Mandatory) : 외래 키에 NULL을 허용하지 않는다. 연관관계를 필수적으로 맺어야 한다.
* 선택적 비식별 관계(Optional) : 외래 키에 NULL을 허용한다. 연관관계를 맺을지 말지 선택할 수 있다.

**7.3.2 복합 키: 비식별 관계 매핑**

-JPA에서 식별자를 둘 이상 사용하려면 별도의 식별자 클래스를 만들어야 한다.

-JPA는 복합 키를 지원하기 위해 @IdClass와 @EmbeddedId 2가지 방법을 제공한다.

-@IdClass는 관계형 데이터베이스에 가까운 방법이고 @EmbeddedId는 좀 더 객체지향에 가까운 방법이다.

**@IdClass(관계형 데이터베이스)**

@Entity  
@IdClass(ParentId.class)  
public class Parent {  
 @Id  
 @Column(name = "PARENT\_ID1")  
 private String id1; //ParentId.id1과 연결  
  
 @Id  
 @Column(name = "PARENT\_ID2")  
 private String id2; //ParentId.id2와 연결  
 private String name;

}

public class ParentId implements Serializable {  
 private String id1; //Parent.id1 매핑  
 private String id2; //Parent.id2 매핑  
  
 public ParentId() {  
  
 }  
  
 public ParentId(String id1, String id2) {  
 this.id1 = id1;  
 this.id2 = id2;  
 }

}

-@IdClass를 사용할 때 식별자 클래스는 다음 조건을 만족해야 한다.

* 식별자 클래스의 속성명과 엔티티에서 사용하는 식별자의 속성명이 같아야 한다.
* Serializable 인터페이스를 구현해야 한다.
* equals, hashCode를 구현해야 한다.
* 기본 생성자가 있어야 한다.
* 식별자 클래스는 public 이어야 한다.

@Entity  
public class Child {  
 @Id  
 private String id;  
  
 @ManyToOne  
 @JoinColumns({  
 @JoinColumn(name = "PARENT\_ID1",  
 referencedColumnName = "PARENT\_ID1"),  
 @JoinColumn(name = "PARENT\_ID2",  
 referencedColumnName = "PARENT\_ID2")  
 })  
 private Parent parent;  
}

-부모 테이블의 기본 키 컬럼이 복합 키이므로 자식 테이블의 외래 키도 복합키다.

**@EmbeddedId(객체지향)**

@Entity  
public class Parent {  
 @EmbeddedId  
 private ParentId id;  
 private String name;

}

@Embeddable  
public class ParentId implements Serializable {  
 @Column(name = "PARENT\_ID1")  
 private String id1;  
  
 @Column(name = "PARENT\_ID2")  
 private String id2;

}

-@IdClass와는 다르게 @EmbeddedId를 적용한 식별자 클래스는 식별자 클래스에 기본 키를 직접 매핑한다.

-@EmbeddedId를 적용한 식별자 클래스는 다음 조건을 만족해야 한다.

* @Embeddable 어노테이션을 붙여주어야 한다.
* Serializable 인터페이스를 구현
* equals, hashCode를 구현
* 기본 생성자가 있어야 한다.
* 식별자 클래스는 public 이어야 한다.

**복합 키와 equals(), hashCode()**

-복합 키는 equals()와 hashCode()를 필수로 구현해야 한다. 보통 모든 필드를 사용

**@IdClass vs @EmbeddedId**

-@EmbeddedId가 @IdClass와 비교해서 더 객체지향적이고 중복도 없어 좋아 보이지만 JPQL이 조금 더 길어질 수 있다.

-복합 키에는 @GenerateValue를 사용할 수 없다.

**7.3.3 복합 키: 식별 관계 매핑**

**@IdClass와 식별 관계**

@Entity  
public class Parent {  
 @Id @Column(name = "PARENT\_ID")  
 private String id;  
 private String name;

…

}

@Entity  
@IdClass(ChildId.class)  
public class Child {  
 @Id  
 @ManyToOne  
 @JoinColumn(name = "PARENT\_ID")  
 public Parent parent;  
  
 @Id  
 @Column(name = "CHILD\_ID")  
 private String childId;

…

}

public class ChildId implements Serializable {  
 private String parent; //Child.parent 매핑  
 private String childId; //Child.childId 매핑

…

}

@Entity  
@IdClass(GrandChildId.class)  
public class GrandChild {  
 @Id  
 @ManyToOne  
 @JoinColumns(  
 {  
 @JoinColumn(name = "PARENT\_ID"),  
 @JoinColumn(name = "CHILD\_ID")  
 }  
 )  
 private Child child;  
  
 @Id @Column(name = "GRANDCHILD\_ID")  
 private String id;  
 private String name;

…

}

public class GrandChildId implements Serializable {  
 private ChildId childId; //GrandChild.child 매핑  
 private String id; //GrandChild.id 매핑

…

}

**@EmbeddedId와 식별 관계**

@Entity  
public class Parent {  
 @Id @Column(name = "PARENT\_ID")  
 private String id;  
 private String name;

…

}

@Entity  
public class Child {  
 @EmbeddedId  
 private ChildId id;  
  
 @MapsId("parentId") //ChildId.parentId 매핑  
 @ManyToOne  
 @JoinColumn(name = " PARENT\_ID")  
 private Parent parent;  
 private String name;

…

}

@Embeddable  
public class ChildId implements Serializable {  
 private String paretId;  
  
 @Column(name = "CHILD\_ID")  
 private String id;

…

}

@Entity  
public class GrandChild {  
 @EmbeddedId  
 private GranChildId id;  
  
 @MapsId("childId") //GrandChildId.childId 매핑  
 @ManyToOne  
 @JoinColumns({  
 @JoinColumn(name = "PARENT\_ID"),  
 @JoinColumn(name = "CHILD\_ID")  
 })  
 private Child child;  
  
 private String name;

…

}

@Embeddable  
public class GranChildId implements Serializable {  
 private ChildId childId; // @MapsId("childId")로 매핑  
  
 @Column(name = "GRANDCHILD\_ID")  
 private String id;

…

}

**7.3.4 비식별 관계로 구현**

-식별 관계의 복합 키를 사용한 코드와 비교하면 매핑도 쉽고 코드도 단순하다. 그리고 복합 키가 없으므로 따로 키 클래스를 만들지 않아도 된다.

**7.3.5 일대일 식별 관계**

@Entity  
public class Board {  
 @Id @GeneratedValue  
 @Column(name = " BOARD\_ID")  
 private Long id;  
  
 private String title;  
  
 @OneToOne(mappedBy = "board")  
 private BoardDetail boardDetail;

}

@Entity  
public class BoardDetail {  
 @Id  
 private Long boardId;  
  
 @MapsId //BoardDetail.boardId 매핑  
 @OneToOne  
 @JoinColumn(name = "BOARD\_ID")  
 private Board board;  
  
 private String content;

}

-자식 테이블의 기본 키 값으로 부모 테이블의 기본 키 값만 사용한다.

**쿼리변환**

Hibernate:  
 create table board (  
 board\_id bigint not null,  
 title varchar(255),  
 primary key ( board\_id)  
 )  
Hibernate:  
 create table board\_detail (  
 board\_id bigint not null,  
 content varchar(255),  
 primary key (board\_id)  
 )  
Hibernate:  
 alter table board\_detail  
 add constraint FK\_ivu7cuf11gnpqoh3m4qfu2q2s  
 foreign key (board\_id)  
 references board  
 Hibernate:  
 create sequence hibernate\_sequence start with 1 increment by 1

**7.3.6 식별, 비식별 관계의 장단점**

-데이터베이스 설계 관점에서 다음과 같은 이유로 식별 관계보다는 비식별 관계를 선호한다.

* 식별 관계는 부모 테이블의 키본 키를 자식 테이블로 전파하면서 자식 테이블의 기본 키 컬럼이 점점 늘어난다. 결국 조인할 때 SQL이 복잡해지고 기본 키 인덱스가 불필요하게 커질 수 있다.
* 식별 관계는 2개 이상의 컬럼을 합해서 복합 키를 만들어야 하는 경우가 많다.
* 식별 관계의 자연 키 컬럼들이 자식에 손자까지 전파되면 변경하기 힘들다.
* 식별 관계는 부모 테이블의 기본 키를 자식 테이블의 기본 키로 사용하므로 비식별 관계보다 테이블 구조가 유연하지 못한다.

-객체 관계 매핑의 관점에서 보면 다음과 같은 이유로 비식별 관계를 선호한다.

* 일대일 관계를 제외하고 식별 관계는 2개 이상의 컬럼을 묶은 복합 기본 키를 사용한다. JPA에서 복합 키는 별도의 복합 키 클래스를 만들어서 사용해야 한다. 따라서 컬럼이 하나인 기본 키를 매핑하는 것보다 많은 노력이 필요하다.
* 비식별 관계의 기본 키는 주로 대리 키를 사용하는데 JPA는 @GenerateValue 처럼 대리 키를 생성하기 위한 편리한 방법을 제공한다.

-식별 관계가 가지는 장점

* 기본 키 엔덱스를 활용하기 좋고, 상위 테이블들의 기본 키 컬럼을 자식, 손자 테이블들이 가지고 있으므로 특정 상황에 조인 없이 하위 테이블만으로 검색을 완료할 수 있음.

**7.4 조인 테이블**

-데이터베이스 테이블의 연관관계 설정 방법

* 조인 컬럼 사용(외래 키)
* 조인 테이블 사용(테이블 사용)

-필수적 비식별 : 외래키에 NULL 허용하지 않는다. (INNER JOIN)

-선택적 비식별 : 외래키에 NULL 허용한다. (OUTER JOIN)

-조인테이블

* 객체와 테이블을 매핑할 때 조인 컬럼은 @JoinColumn으로 매핑하고 조인 테이블은 @JoinTable로 매핑한다.
* 조인 테이블은 주로 다대다 관계를 일대다, 다대일 관계로 풀어내기 위해 사용한다.

**7.4.1 일대일 조인 테이블**

-일대일 관계를 만들려면 조인 테이블의 외래 키 컬럼에 각각에 총 2개의 유니크 제약조건을 걸어야 한다.

-@JoinTable의 속성

* name: 매핑할 조인 테이블 이름
* joinColumns: 현재 엔티티를 참조하는 외래 키
* inverseJoinColumns: 반대방향 엔티티를 참조하는 외래 키

**7.4.2 일대다 조인 테이블**

-일대다 관계를 만들려면 조인 테이블의 컬럼 중 다(N)와 관련된 컬럼에 유니크 제약 조건을 걸어야 한다.

**7.4.3 다대일 조인 테이블**

-다대일은 일대다에서 방향만 반대이다.

**7.4.4 다대다 조인 테이블**

-다대다 관계를 만들려면 조인 테이블의 두 컬럼을 합해서 하나의 복합 유니크 제약조건을 걸어야 한다.

**7.5 엔티티 하나에 여러 테이블 매핑**

-잘 사용하지 않지만 @SecondaryTable을 사용하면 한 엔티티에 여러 테이블을 매핑할 수 있다.

@Entity  
@Table(name = "BOARD")  
@SecondaryTable(name = "BOARD\_DETAIL",  
 pkJoinColumns = @PrimaryKeyJoinColumn(name = "BOARD\_DETAIL\_ID"))  
public class Board {  
 @Id @GeneratedValue  
 @Column(name = "BOARD\_ID")  
 private Long id;  
  
 private String title;  
  
 @Column(table = "BOARD\_DETAIL")  
 private String content;

}

[-@SecondaryTable.name](mailto:-@SecondaryTable.name) : 매핑할 다른 테이블의 이름

[-@SecondaryTable.pkJoinColumns](mailto:-@SecondaryTable.pkJoinColumns): 매핑할 다른 테이블의 기본 키 컬럼 속성

-@Column(table = “”): 특정 테이블에 컬럼 매핑. 위 속성을 지정해 주지 않으면 기본 테이블로 매핑

-참고로 위 방법보다 테이블당 엔티티를 각각 만들어서 일대일 매핑하는 것을 권장

**CHAPTER8**

**프록시와 즉시로딩, 지연로딩**

-객체는 객체 그래프로 연관된 객체들을 탐색한다.

그런데 객체가 데이터베이스에 저장되어 있으므로 연관된 객체를 마음껏 탐색하기 어렵다.

JPA는 이 문제를 해결하려고 프록시라는 기술을 사용한다.

-프록시를 사용하면 연관된 객체를 처음부터 데이터베이스에서 조회하는 것이 아니라, 실제 사용하는 시점에 조회할 수 있다.

-하지만 자주 사용하는 객체들은 조인을 사용해서 함께 조회하는 것이 효과적이다.

**영속성 전이와 고아 객체**

-JPA는 연관된 객체를 함께 저장하거나 삭제할 수 있는 편리한 기능을 제공한다.

**8.1 프록시**

-엔티티를 조회할 때 연관된 엔티티들이 항상 사용되는 것은 아니다.

-지연 로딩 기능을 사용하려면 실제 엔티티 객체 대신에 데이터베이스 조회를 지연할 수 있는 가짜 객체가 필요한데 이것을 프록시 객체라 한다.

**참고**

-JPA 표준 명세는 지연 로딩의 구현 방법을 JPA 구현체에 위임했다.

**8.1.1 프록시 기초**

Member memberA = em.getReference(Member.class, "memberA");

🡺getReference를 사용하면 실제 사용하는 시점까지 조회를 미룰 수 있다.

- 이 메소드를 호출할 때 JPA는 데이터베이스를 조회하지 않고 실제 엔티티 객체도 생성하지 않는다. 다만 데이터베이스 접근을 위임한 프록시 객체를 반환한다.

-프록시의 특징

* 프록시 객체는 처음 사용할 때 한 번만 초기화된다.
* 프록시 객체를 초기화 한다고 실제 엔티티로 변경되는 것은 아니다. 다만 프록시를 통해서 실제 엔티티에 접근할 수 있다.
* 프록시 객체는 원본 엔티티를 상속받은 객체이므로 타입 체크시에 주의해야 한다.
* 영속성 컨텍스트에 찾는 엔티티가 이미 있으면 em.getReference()를 호출해도 프록시가 아닌 실제 엔티티를 반환한다.
* 초기화는 영속성 컨텍스트의 도움을 받아야 가능하다.
* 준영속성 상태의 프록시를 초기화 하면 LazyInitializationExecption 발생

**8.1.2 프록시와 식별자**

-엔티티를 프록시로 조회할 떄 식별자(PK) 값을 파라미터로 전달하는데 프록시 객체는 이 식별자 값을 보관한다.

Member memberA = em.getReference(Member.class, "memberA");  
memberA.getId();

🡺프록시 객체가 식별자를 가지고 있기 때문에 getId를 호출해도 초기화하지 않는다. (@Access(AccessType.PROPERTY)) 를 사용한 경우에만

**8.1.3 프록시 확인**

-JPA가 제공하는 PersistenceUnitUtil.isLoaded 메소드를 사용하면 프록시 인스턴스 초기화 여부를 확인할 수 있다. (이미 초기화 되었으면 true)

**8.2. 즉시 로딩과 지연 로딩**

**즉시 로딩**

-즉시 로딩 : 엔티티를 조회할 때 연관된 엔티티도 함께 조회한다.

ex) em.find(Member.class, “member”1)를 호출 할 때 연관된 팀 엔티티도 함꼐 조회

-설정 방법 : @ManyToOne(fetch = FetchType.EAGER)

**지연 로딩**

-지연 로딩 : 연관된 엔티티를 실제 사용할 때 조회한다.

ex) member.getTeam().getName() 처럼 조회한 팀 엔티티를 실제 사용하는 시점에 조회

-설정 방법 : @ManyToOne(fetch = FetchType.LAZY)

**8.2.1 즉시 로딩**

@ManyToOne(fetch = FetchType.*EAGER*)  
@JoinColumn(name = "TEAM\_ID")  
private Team team;

-대부분의 JPA 구현체는 즉시 로딩을 최적화하기 위해 가능하면 조인 쿼리를 사용한다.

**참고**

-@JoinColumn에 nullable = false를 설정해서 이 외래 키는 NULL 값을 허용하지 않는다고 알려주면 JPA는 외부 조인 대신에 내부 조인을 사용한다.

@ManyToOne  
@JoinColumn(name = "TEAM\_ID", nullable = false)  
private Team team;

또는

@ManyToOne(fetch = FetchType.*EAGER*, optional = false)  
@JoinColumn(name = "TEAM\_ID")  
private Team team;

**8.2.2 지연 로딩**

@ManyToOne(fetch = FetchType.*LAZY*)  
@JoinColumn(name = "TEAM\_ID")  
private Team team;

-조회 대상이 영속성 컨텍스트에 이미 있으면 프록시 객체를 사용할 이유가 없다.

**8.3 지연 로딩 활용**

**8.3.1 프록시와 컬렉션 래퍼**

Member member = em.find(Member.class, "member1");  
List<Order> orders = member.getOrders();

🡺orders.getClass().getName() 정보를 보면 org.hibernate.colllection.internal.PersistentBag 로 출력되며 이는 컬렉션을 추적하고 관리할 목적으로 하이버네이트가 제공하는 내장 컬렉션으로 변경하는데 이것을 컬렉션 레퍼라고 한다.

-엔티티 지연로딩 🡺 프록시 객체가 처리

-컬렉션 지연로딩 🡺 컬렉션 래퍼가 처리

-참고로 member.getOrders() 처럼 호출해도 컬렉션 초기화가 되지 않는다. member.getOrders().get(0) 처럼 실제 데이터를 조회할 때 초기화 실행

**8.3.2 JPA 기본 페치 전략**

@ManyToOne(fetch = FetchType.*EAGER*)  
@OneToMany(fetch = FetchType.*LAZY*)

-연관된 엔티티가 하나면 즉시 로딩을, 컬렉션이면 지연 로딩을 사용한다.

-추천하는 방법은 모든 연관관계에 지연 로딩을 사용하는 것이다.

-SQL은 이런 페치 전략으로 변경하기 어렵다. (변경점 많다.)

**8.3.3 컬렉션에 FetchType.EAGER 사용 시 주의점**

-컬렉션을 하나 이상 즉시 로딩하는 것은 권장하지 않는다.

* 일대다 조인은 결과 데이터가 다 쪽에 있는 수만큼 증가하게 된다.
* 문제는 서로 다른 컬렉션을 2개 이상 조인할 때 N\*M이 되면서 많은 데이터가 조회 되고 성능이 저하될 수 있다.

-컬렉션 즉시 로딩은 항상 외부 조인(OUTER JOIN)을 사용한다.

**FetchType.EAGER 설정과 조인 전략**

-@ManyToOne, @OneToOne

* (optional = false) : 내부 조인
* (optional = true) : 외부 조인

-@OneToMany, @ManyToMany

* (optional = false) : 외부 조인
* (optional = true) : 외부 조인

**8.4 영속성 전이: CASCADE**

-특정 엔티티를 영속 상태로 만들 때 연관된 엔티티도 함께 영속 상태로 만들고 싶으면 영속성 전이 기능을 사용하면 된다.

private static void saveNoCascade(EntityManager em) {  
 //부모 저장  
 Parent parent = new Parent();  
 em.persist(parent);  
  
 //1번 자식 저장  
 Child child1 = new Child();  
 child1.setParent(parent); //자식 -> 부모 연관관계 설정  
 parent.getChildren().add(child1); //부모 -> 자식  
 em.persist(child1);  
  
 //2번 자식 저장  
 Child child2 = new Child();  
 child2.setParent(parent); //자식 -> 부모 연관관계 설정  
 parent.getChildren().add(child2); //부모 -> 자식  
 em.persist(child2);  
}

🡺영속성 상태를 만들기 위해 각각 다 persist를 이용해서 저장 했다.

**8.4.1 영속성 전이: 저장**

@Entity  
public class Parent {  
 @Id @GeneratedValue  
 private Long id;  
  
 @OneToMany(mappedBy = "parent", cascade = CascadeType.*PERSIST*)  
 private List<Child> children = new ArrayList<Child>();

}

private static void saveWithCascade(EntityManager em) {  
 Child child1 = new Child();  
 Child child2 = new Child();  
  
 Parent parent = new Parent();  
 child1.setParent(parent); //연관관계 추가  
 child2.setParent(parent); //연관관계 추가  
 parent.getChildren().add(child1);  
 parent.getChildren().add(child2);  
  
 //부모 저장, 연관된 자식드롣 저장  
 em.persist(parent);  
}

-CASCADE 옵션을 적용해서 부모를 영속화할 때 연관된 자식들도 함께 영속화할 수 있다.

**8.4.2 영속성 전이: 삭제**

private static void remove(EntityManager em) {  
 Parent findParent = em.find(Parent.class, 1L);  
 Child findChild1 = em.find(Child.class, 1L);  
 Child findChild2 = em.find(Child.class, 2L);  
  
 em.remove(findChild1);  
 em.remove(findChild2);  
 em.remove(findParent);  
}

-부모와 자식 엔티티를 모두 제거하려면 다음 코드와 같이 엔티티를 하나씩 제거해야 한다.

@Entity  
public class Parent {  
 @Id @GeneratedValue  
 private Long id;  
  
 @OneToMany(mappedBy = "parent", cascade = CascadeType.*REMOVE*)  
 private List<Child> children = new ArrayList<Child>();

}

private static void remove(EntityManager em) {  
 Parent findParent = em.find(Parent.class, 1L);  
 em.remove(findParent);  
}

-CascadeType.REMOVE를 사용하면 연관 엔티티 모두 삭제할 수 있다.

**8.4.3 CASCADE의 종류**

public enum CascadeType { *ALL*, // 모두 적용 *PERSIST*, // 영속 *MERGE,* // 병합

*REMOVE*, // 삭제 *REFRESH*, // REFRESH *DETACH* // DETACH

}

-cascade = {CascadeType.PERSIST, CascadeType.REMOVE} 같이 여러 속성 가능

**8.5 고아 객체**

-JPA는 부모 엔티티와 연관관계가 끊어진 자식 엔티티를 자동으로 삭제하는 기능을 제공하는데 이것을 고아 객체 제거라 한다.

@Entity  
public class Parent {  
 @Id @GeneratedValue  
 private Long id;  
  
 @OneToMany(mappedBy = "parent", orphanRemoval = true)  
 private List<Child> children = new ArrayList<Child>();

}

private static void orphanRemove(EntityManager em) {  
 Parent parent1 = em.find(Parent.class, 1L);  
 parent1.getChildren().clear(); // 왜 안될까?  
}

-참조가 제거된 엔티티는 다른 곳에서 참조하지 않는 고아 객체로 보고 삭제한다.

**8.6 영속성 전이 + 고아 객체, 생명주기**

-CascadeType.ALL + orphanRemoval = true를 동시에 사용하면?

🡺부모 엔티티를 통해서 자식의 생명 주기를 관리할 수 있다.

// 저장  
Parent parent = em.find(Parent.class, parentId);  
parent.addChild(child1);  
  
// 삭제  
Parent parent = em.find(Parent.class, parentId);  
parent.getChildren().remove(removeObject);

**CHAPTER9**

-엔티티 타입은 식별자를 통해 지속해서 추젖ㄱ할 수 있지만, 값 타입은 식별자가 없고 숫자나 문자같은 속성만 있으므로 추적할 수 없다.

-기본값 타입(basic value type)

* 자바 기본 타입(int, double)
* 래퍼 클래스(Integer)
* String

-임베디드 타입(embedded type)

* 복합 값 타입

-컬렉션 값 타입(collection value type)

**9.1 기본값 타입**

@Entity  
public class Member {  
 @Id @GeneratedValue  
 private Long id;  
 private String name;  
 private int age;

}

-String, int가 값 타입이며 식별자 값도 없고 생명주기도 회원 엔티티에 의존한다.

**9.2 임베디드 타입(복합 값 타입)**

@Entity  
public class Member {  
 @Id @GeneratedValue  
 private Long id;  
 private String name;  
  
 @Embedded Period workPeriod; //근무 기간  
 @Embedded Address homeAddress; //집 주소

}

@Embeddable  
public class Period {  
 //근무 기간  
 @Temporal(TemporalType.*DATE*) Date startDate;  
 @Temporal(TemporalType.*DATE*) Date endDate;

public boolean isWork(Date date) {  
 return true;  
 }

}

@Embeddable  
public class Address {  
 @Column(name = "city")  
 private String city;  
 private String street;  
 private String zipcode;

}

-엔티티가 의미 있고 응집력 있다.

-Period.isWork() 처럼 해당 값 타입만 사용하는 의미 있는 메소들을 만들 수 있다. (DDD)

-임베디드 타입은 기본 생성자가 필수다.

-모든 값 타입은 엔티티의 생명주기에 의존하므로 **컴포지션** 관계가 된다.

-@Embeddable : 값 타입을 정의하는 곳에 표시

-@Embedded : 값 타입을 사용하는 곳에 표시

**9.2.1 임베디드 타입과 테이블 매핑**

-값 타입과 임베디드 타입의 테이블은 같다.

-임베디드 타입 덕분에 객체와 테이블을 아주 세밀하게 매핑하는 것이 가능 (클래스 수가 테이블보다 더 많아짐)

**9.2.2 임베디드 타입과 연관관계**

-임베디드 타입은 값 타입을 포함하거나 엔티티를 참조할 수 있다.

@Entity  
public class Member {  
 @Id @GeneratedValue  
 private Long id;  
  
 @Embedded Address address; //임베디드 타입 포함  
 @Embedded PhoneNumber phoneNumber; //임베디드 타입 포함  
}

@Embeddable  
public class PhoneNumber {  
 String areaCode;  
 String localNumber;  
 @ManyToOne PhoneServiceProvider provider; //엔티티 참조  
}

@Entity  
public class PhoneServiceProvider {  
 @Id String name;  
}

**9.2.3 @AttributeOverride: 속성 재정의**

@Entity  
public class Member {  
 @Id @GeneratedValue  
 private Long id;  
 private String name;  
  
 @Embedded Address homeAddress;  
 @Embedded Address companyAddress;  
}

🡺Address 컬럼이 중복되었다.

@Entity  
public class Member {  
 @Id @GeneratedValue  
 private Long id;  
 private String name;  
  
 @Embedded Address homeAddress;  
  
 @Embedded  
 @AttributeOverrides({  
 @AttributeOverride(name = "city", column = @Column(name = "COMPANY\_CITY")),  
 @AttributeOverride(name = "street", column = @Column(name = "COMPANY\_STREET")),  
 @AttributeOverride(name = "zipcode", column = @Column(name = "COMPANY\_ZIPCODE"))  
 })  
 Address companyAddress;  
}

-@AttributeOverride를 사용하면 어노테이션을 너무 많이 사용해서 엔티티 코드가 지저분해진다. 다행히 보통 타입을 중복해서 사용하는 일은 많지 않다.

**9.2.4 임베디드 타입과 null**

member.setAddress(null);  
em.persist(member);

🡺임베디드 타입이 null이면 매핑한 컬럼 값은 모두 null이 된다.

**9.3 값 타입과 불변 객체**

**9.3.1 값 타입 공유 참조**

-값 타입을 공유하면 부작용이 발생하며 이를 막으려면 값을 복사해서 사용하면 된다.

**9.3.2 값 타입 복사**

-공유 참조로 인해 발생하는 부작용을 피할 수 있지만, 임베디드 타입처럼 직접 정의한 값 타입은 자반의 기본 타입이 아니라 객체 타입이라는 것이다.

-문제는 복사하지 않고 원본의 참조 값을 직접 넘기는 것을 막을 방법이 없다.

-근본적인 객체의 공유 참조를 피할 수 없다.

**9.3.3 불변 객체**

-객체를 불변하게 만들면 값을 수정할 수 없으므로 부작용을 원천 차단할 수 있다.

-값 타입은 되도록 불변 객체로 설계해야 한다.

@Embeddable  
public class Address {  
 private String city;  
  
 protected Address() {  
 } // JPA에서 기본 생성자는 필수  
  
 // 생성자로 초기 값을 설정한다.  
 public Address(String city) {  
 this.city = city;  
 }  
  
 // 접근자(Getter)는 노출한다.  
 public String getCity() {  
 return city;  
 }  
}

🡺생성자로만 값을 설정하고 수정자(setter)를 만들지 않으면 된다.

**9.4 값 타입 비교**

-동일성 비교 : 인스턴스의 참조 값을 비교 (==)

-동등성 비교 : 인스턴스의 값을 비교 (equals)

**참고**

-자바에서 equals()를 재정의하면 hashCode()도 재정의하는 것이 안전하다. 그렇지 않으면 해시를 사용하는 컬렉션 (HashSet, HashMap)이 정상 동작하지 않는다.

**9.5 값 타입 컬렉션**

@Entity  
public class Member {  
 @Id @GeneratedValue  
 private Long id;  
  
 @Embedded  
 private Address homeAddress;  
  
 @ElementCollection  
 @CollectionTable(name = "FAVORITE\_FOODS",  
 joinColumns = @JoinColumn(name = "MEMBER\_ID"))  
 @Column(name = "FOOD\_NAME")  
 private Set<String> favoriteFoods = new HashSet<String>();  
  
 @ElementCollection  
 @CollectionTable(name = "ADDRESS",  
 joinColumns = @JoinColumn(name = "MEMBER\_ID"))  
 private List<Address> addressesHistory = new ArrayList<Address>();

}

@Embeddable  
public class Address {  
 @Column  
 private String city;  
 private String street;  
 private String zipcode;

}

-값 타입을 하나 이상 저장하려면 컬렉션에 보관하고 @ElementCollection, @CollectionTable 어노테이션을 사용하면 된다.

**9.5.1 값 타입 컬렉션 사용**

Member member = new Member();  
  
//임베디드 값 타입  
member.setHomeAddress(new Address("통영", "몽돌해수욕장", "660-123"));  
  
//기본값 타입 컬렉션  
member.getFavoriteFoods().add("짬뽕");  
member.getFavoriteFoods().add("짜장");  
member.getFavoriteFoods().add("탕수육");  
  
//임베디드 값 타입 컬렉션  
member.getAddressesHistory().add(new Address("서울", "강남", "123-123"));  
member.getAddressesHistory().add(new Address("서울", "강복", "000-000"));  
em.persist(member);

-em.persist 한 번 호출로 총 6번의 INSERT SQL을 실행한다.

**참고**

-값 타입 컬렉션은 영속성 전이(Cascade) + 고아 객체 제거(ORPHAN REMOVE) 기능을 필수로 가진다.

**9.5.2 값 타입 컬렉션의 제약사항**

-엔티티는 식별자가 있으므로 데이터베이스에 저장된 우너본 데이터를 쉽게 찾아서 변경할 수 있다.

-반면 값 타입은 식별자라는 개념이 없고 단순한 값들의 모음이므로 값을 변경해버리면 워본 데이터를 찾기 어렵다.

-값 타입 컬렉션에 변경 사항이 발생하면, 값 타입 컬렉션이 매핑된 테이블의 연관된 모든 데이터를 삭제하고, 현재 값 타입 컬랙션 객체에 있는 모든 값을 데이터 베이스에 다시 저장한다.

🡺따라서 실무에서는 값 타입 컬렉션이 매핑된 테이블에 데이터가 많다면 값 타입 컬렉션 대신에(삭제 하고 다시 저장 하므로) 일대다 관계를 고려해야 한다.

**CHAPTER10**

**10.1 객체지향 쿼리 소개**

-메서드 쿼리는 한계성이 있음. 모든 데이터를 메모리에 올려서 처리할 것인가?

-데이터베이스 테이블이 아닌 엔티티 객체를 대상으로 검색 (JPQL 탄생)

**JPQL 특징**

-테이블이 아닌 객체를 대상으로 검색하는 객체지향 쿼리

-SQL을 추상화해서 특정 데이터베이스 SQL에 의존하지 않는다.

**JPA가 공식 지원**

-JPQL (Java Persistence Query Language)

-Criteria 쿼리 (Criteria Query) : JPQL을 편하게 작성하도록 도와주는 API, 빌더 클래스 모음

-네이티브 SQL (Native SQL) : JPA에서 JPQL 대신 직접 SQL을 사용할 수 있다.

**JPA 공식 지원 외**

-QueryDSL : Criteria 쿼리처럼 JPQL을 편하게 작성하도록 도와주는 빌더 클래스. 비표준 오픈소스 프레임워크

-JDBC 직접 사용, MyBatis 같은 SQL 매퍼 프레임워크 사용

-결국 Criteria나 QueryDSL은 JPQL을 편하게 작성하도록 도와 주는 빌더이기 때문에 JPQL이 가장 중요 한다.

**10.1.1 JPQL 소개**

-JPQL은 엔티티 객체를 조회하는 객체지향 쿼리다. (SQL과 비슷)

-JPQL은 SQL을 추상화해서 특정 데이터베이스에 의존하지 않는다.

-JPQL은 SQL보다 간결하다. (엔티티 직접 조회, 묵시적 조인, 다형성 지원 등)

@Entity  
public class Member {  
 @Column(name = "NAME", nullable = false, length = 10)  
 private String username;

…

}

List<Member> members = em.createQuery("select m from Member as m where m.username = 'kim'", Member.class).getResultList();

🡺m.username은 테이블 컬럼명이 아니라 엔티티 객체의 필드명이다.

🡺JPA는 JPQL을 SQL로 변환해서 데이터베이스를 조회한다. 그리고 결과를 Member 엔티티를 생성해서 반환한다.

**10.1.2 Criteria 쿼리 소개**

-JPQL을 생성하는 빌더 클래스로 장점은 문자가 아닌 query.select(m).where(…) 처럼 프로그래밍 코드로 JPQL을 작성할 수 있다.

-JPQL의 단점은 오타를 컴파일 시점에서 확인할 수 없다.

**Criteria 장점**

* 컴파일 시점에서 오류 발견
* IDE를 사용하면 코드 자동완성 지원
* 동적 쿼리를 작성하기 편함

//Criteria 사용 준비  
CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Member> query = cb.createQuery(Member.class);  
  
//루트 클래스(조회를 시작할 클래스)  
Root<Member> m = query.from(Member.class);  
  
//쿼리 생성  
CriteriaQuery<Member> cq =  
 query.select(m).where(cb.equal(m.get("username"), "kim"));  
List<Member> resultList = em.createQuery(cq).getResultList();

**메타모델**

-어노테이션을 분석해서 클래스를 생성할 수 있다.

ex) Member 엔티티 🡺 Member\_ Criteria 전용 클래스

-Criteria가 가진 장점이 많지만 복잡하고 장황하다. 사용하기 불편하다.

**10.1.3 QueryDSL 소개**

-Criteria 처럼 JPQL 빌더 역할을 한다.

-코드 기반이며 단순하고 사용하기 쉽다.

**pom.xml**

<!-- querydsl -->  
<dependency>  
 <groupId>com.querydsl</groupId>  
 <artifactId>querydsl-apt</artifactId>  
 <version>${querydsl.version}</version>  
</dependency>  
<dependency>  
 <groupId>com.querydsl</groupId>  
 <artifactId>querydsl-jpa</artifactId>  
 <version>${querydsl.version}</version>  
</dependency>

<build>  
 <plugins>  
 <plugin>  
 <groupId>com.mysema.maven</groupId>  
 <artifactId>apt-maven-plugin</artifactId>  
 <version>1.1.3</version>  
 <executions>  
 <execution>  
 <goals>  
 <goal>process</goal>  
 </goals>  
 <configuration>  
 <outputDirectory>target/generated-sources/java</outputDirectory>  
 <processor>com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

**JPAQueryFactory**

JPAQueryFactory query = new JPAQueryFactory(em);  
QMember qMember = QMember.*member*;  
  
//쿼리, 결과 조회  
List<Member> members =  
 query.selectFrom(qMember)  
 .where(qMember.username.eq("kim"))  
 .fetch();

-3.X 에는 JPAQuery를 사용한 것 같은데 4.X 부터 JPAQueryFactory로 변경된 것 같다.

책에 있는 예제 실행 안됨. (좀 해맸음)

**Intellij**



-pom.xml plugins에 설정되어 있는 srud로가 Source folders 로 잡혀 있어야 Q파일을 인식할 수 있다.

**10.1.4 네이티브 SQL 소개**

-JPA는 SQL을 직접 사용할 수 있는 기능을 지원한다. 이를 네이티브 SQL

-JQAL에서 지원하지 않은 데이터베이스 방언은 네이티브 SQL을 이용하면 된다.

-단점은 특정 데이터베이스에 의존하기 때문에 변경되면 네이티브 SQL도 수정해야 된다.

//조회  
String sql = "SELECT ID, AGE, NAME FROM MEMBER WHERE NAME = 'kim'";  
List<Member> resultList = em.createNativeQuery(sql, Member.class).getResultList();

결과

Hibernate:  
 /\* dynamic native SQL query \*/ SELECT  
 ID,  
 AGE,  
 NAME  
 FROM  
 MEMBER  
 WHERE  
 NAME = 'kim'

**10.1.5 JDBC 직접 사용, 마이바티스 같은 SQL 매퍼 프레임워크 이용**

-JDBC나 마이바티스를 JPA와 함께 사용하면 영속성 컨텍스트를 적절한 시점에 강제로 플러시해야 한다. (JPA를 우회하기 때문에…)

**10.2 JPQL**

-JPQL은 객체지향 쿼리 언어다. 따라서 엔티티 객체를 대상으로 쿼리한다.

-JPQL은 SQL을 추상화해서 특정 데이터베이스 SQL에 의존하지 않는다.

-JPQL은 결국 SQL로 변환된다.

**10.2.1 기본 문법과 쿼리 API**

**SELECT 문**

SELECET m FROM Member AS m where m.username = ‘Hello’

**특징**

-대소문자 구문 : 엔티티와 속성은 대소문자를 구분. 반면 SELECT, FROM, AS 같은 JPQL 키워드는 대소문자를 구분하지 않는다.

-엔티티 이름 : 클래스 명이라 엔티티 명이다. ex) @Entity(name = “XXX”), 엔티티명을 지정하지 않으면 클래스명을 기본값으로 한다.

-별칭은 필수 : Member AS m 같이 m이라는 별칭을 주었다. JPQL은 별칭을 필수

**TypeQuery, Query**

-작성한 JPQL을 실행하려면 쿼리 객체를 만들어야 한다. (TypeQuery, Query)

-반환할 타입을 명확하게 지정할 수 있으면 TypeQuery 객체를 사용

-반환할 타입을 명확하게 지정할 수 없으면 Query 객체를 사용

**TypedQuery**

TypedQuery<Member> query = em.createQuery("SELECT m FROM Member m", Member.class);  
  
List<Member> resultList = query.getResultList();  
for (Member mem : resultList) {  
 System.*out*.println("member = " + mem);  
}

**Query**

Query query = em.createQuery("SELECT m.username, m.age from Member m");  
List resultList = query.getResultList();  
  
for (Object o : resultList) {  
 Object[] result = (Object[]) o;  
 System.*out*.println("username = " + result[0]);  
 System.*out*.println("age = " + result[1]);  
}

🡺엔티티나 컬럼을 선택할 때는 반환할 타입을 명확하지 않으므로 Query 객체 사용

**결과 조회**

-query.getResultList() : 결과를 예제로 반환

-query.getSingleResult() : 결과가 정확히 하나일 때 사용

* 결과가 없으면 javax.persistence.NoResultException 예외
* 결과가 1개보다 많으면 javax.persistence.NonUniqueResultException 예외

**10.2.2 파라미터 바인딩**

-JDBC는 위치 기준 파라미터 바인딩만 지원하지만 JPQL은 이름 기준 파라미터 바인딩도 지원한다.

**이름 기준 파라미터**

String usernameParam = "kim";  
  
TypedQuery<Member> query =  
 em.createQuery("SELECT m FROM Member m where m.username = :username", Member.class);  
  
query.setParameter("username", usernameParam);  
List<Member> resultList = query.getResultList();

🡺이름 기준 파라미터는 앞에 : 를 사용한다.

**위치 기준 파라미터**

String usernameParam = "kim";  
  
List<Member> members =  
 em.createQuery("SELECT m FROM Member m where m.username = ?1", Member.class)  
 .setParameter(1, usernameParam)  
 .getResultList();

**참고**

“select m from Member m where m.username =’” + usernameParam + “’”

🡺이런 방식은 SQL 인젝션에도 위험하고, 성능 이슈도 생길 수 있다. (걍 바인딩 써라)

**10.2.3 프로젝션**

-SELECT 절에 조회할 대상을 지정하는 것을 프로젝션(projection)이라 하고 [SELECT {프로젝션 대상} FROM] 으로 대상을 선택한다.

-종류로는 엔티티, 엠비디드 타입, 스칼라 타입 등이 있다.

**엔티티 프로젝션**

SELECT m FROM Member m // 회원  
SELECT m.team FROM Member m // 팀

🡺이렇게 조회한 엔티티는 영속성 컨텍스트에서 관리된다.

**임베디드 타입 프로젝션 (ex @Embedded Address address)**

List<Address> addresses = em.createQuery("SELECT o.address FROM Order o", Address.class).getResultList();

🡺엔티티와 비슷하지만 조회의 시작점이 될 수 없다.

🡺임베디드 타입은 엔티티 타입이 아닌 값 타입이다. 영속성 컨텍스트에서 관리되지 않는다.

**스칼라 타입 프로젝션**

List<String> usernames = em.createQuery("SELECT username FROM Member m", String.class).getResultList();

🡺숫자, 문자, 날짜와 같은 기본 데이터 타입들을 스칼라 타입이라고 한다.

🡺통계 쿼리도 주로 스칼라 타입 조회 (ex AVG, SUM)

**여러 값 조회**

-Query 사용

**NEW 명령어**

TypedQuery<UserDTO> query = em.createQuery("SELECT new chapter10.jpql.typequery.UserDTO(m.username, m.age) FROM Member m",  
 UserDTO.class);  
  
List<UserDTO> resultList = query.getResultList();

🡺new 명령어를 사용하면 TypeQuery로 넘길 수 있어서 지루한 객체 변환 작업을 줄일 수 있다.

**주의점**

-패키지 명을 포함한 전체 클래스 명을 입력해야 한다.

-순서와 타입이 일치하는 생성자가 필요하다.

**10.2.4 페이징 API**

-JPA는 페이징을 다음 두 API로 추상화 했다.

* setFirstResult(int startPosition) : 조회 시작 위치
* setMaxResult(int maxResult) : 조회할 데이터 수

TypedQuery<Member> query = em.createQuery("SELECT m FROM Member m ORDER BY m.username DESC", Member.class);  
  
query.setFirstResult(10);  
query.setMaxResults(20);  
List<Member> resultList = query.getResultList();

🡺11~30번 데이터를 조회 한다.

**10.2.5 집합과 정렬**

-집합함수와 함께 통계 정보를 구할 때 사용한다.

**집합 함수**

|  |  |
| --- | --- |
| 함수 | 설명 |
| COUNT | 결과 수를 구한다. 반환 타입: Long |
| MAX, MIN | 최대, 최소 값을 구한다. 문자, 숫자, 날짜 등에 사용한다. |
| AVG | 평균값을 구한다. 숫자타입만 사용할 수 있다. 반환 타입 : Double |
| SUM | 합을 구한다. 숫자타입만 사용할 수 있다.  반환 타입 : 정수합 Long,  소수합: Double  Biginteger합 : Biginteger,  BigDecimal합 : BigDecimal |

**집합 함수 사용 시 참고**

-NULL 값은 무시하므로 통계에 잡히지 않는다.

-만약 값이 없는데 SUM, AVG, MAX, MIN 함수를 사용하면 NULL 값이 된다. 단 count는 0

-DISTINCT를 집합 함수 안에 사용해서 중복된 값을 제거하고 나서 집합을 구할 수 있다.

ex) select COUNT (DISTINCT m.age ) from Member m

-DISTINCT를 COUNT에서 사용할 때 임베디드 타입은 지원하지 않는다.

**GROUP BY, HAVING**

-GROUP BY는 통계 데이터를 구할 때 특정 그룹끼리 묶어준다.

-HAVING은 GROUP BY와 함께 사용하는데 GROUP BY로 그룹화한 통계 데이터 기준으로 필터링

🡺이런 쿼리들을 보통 리포틸 쿼리나 통계 쿼리라고 한다. (보통 전체 데이터를 기준으로 처리하기 때문에 실시간으로 사용하기에는 부담이 많다.)

**정렬(ORDER BY)**

-ASC : 오름차순

-DESC : 내림차순

**10.2.6 JPQL 조인**

**내부 조인**

String teamName = "team1";  
String query = "SELECT m FROM Member m INNER JOIN m.team t "  
 + "WHERE t.name = :teamName";  
  
// Member 기준 Entity Join (Team 값이 없다.)  
List<Member> members = em.createQuery(query, Member.class)  
 .setParameter("teamName", teamName)  
 .getResultList();  
  
String query2 = "SELECT m, t FROM Member m JOIN m.team t";  
// 두 엔티티 모두 조회  
List<Object[]> result = em.createQuery(query2).getResultList();  
for (Object[] row : result) {  
 Member member = (Member) row[0];  
 Team team = (Team) row[1];  
}

🡺INNER는 생략 가능

-JPQL 조인의 가장 큰 특징은 연관 필드를 사용한다는 것이다. (다른 엔티티와 연관관계를 가지기 위해 사용하는 필드)

ex) FROM Member m JOIN Team t (X) // 잘못된 JPQL 조인으로 오류 난다.

FROM Member m JOIN m.team t (O)

**외부 조인**

String username = "kim2";  
String query = "SELECT m FROM Member m LEFT OUTER JOIN m.team t "  
 + "WHERE m.username = :username";  
  
List<Member> members = em.createQuery(query, Member.class)  
 .setParameter("username", username)  
 .getResultList();

🡺OUTER는 생략 가능해서 보통 LEFT JOIN 으로 사용

**컬렉션 조인**

-일대다 관계나 다대다 관계처럼 컬렉션을 사용하는 곳에 조인하는 것을 컬렉션 조인

ex)

* [회원 -> 팀]으로 조인은 다대일 조인이면서 단일 값 연관 필드(m.team)를 사용
* [팀 -> 회원]은 반대로 일대다 조인이면서 컬렉션 값 연관 필드(m.members)를 사용

**세타 조인**

-WHERE 적을 사용해서 세타 조인을 할 수 있으며, 내부 조인만 지원한다.

-관계없는 엔티티도 조인할 수 있다.

String query = "select count(m) from Member m, Team t where m.username = t.name";  
Long result = em.createQuery(query, Long.class).getSingleResult();

🡺연관관계가 없는 Member의 username과 Team의 name을 join 했음.

**JOIN ON 절**

-ON 절을 사용하면 조인 대상을 필터링 하고 조인할 수 있다.

-내부 조인의 ON 절은 WHERE 절을 사용할 떄와 결과가 같으므로 보통 외부 조인에서만 사용한다.

String query = "select m, t from Member m left join m.team t on t.name = 'team2'";  
List<Object[]> result = em.createQuery(query).getResultList();  
for (Object[] row : result) {  
 Member member = (Member) row[0];  
 Team team = (Team) row[1];  
}

SQL

select  
m.\*, t.\*  
 from  
member m  
left join team t  
on m.team\_id = t.team\_id  
and t.name = 'team1';

결과 (team1로 필터)



**10.2.7 페치 조인**

-SQL 조인이 아니고 JPQL에서 성능 최적화를 위해 제공하는 기능이다.

-연관된 엔티티나 컬렉션을 한 번에 같이 조회 하는 기능이다.

**엔티티 페치 조인**

String jpql = "select m from Member m join fetch m.team";  
  
List<Member> members = em.createQuery(jpql, Member.class)  
 .getResultList();  
  
for (Member member : members) {  
 // 페치 조인으로 회원과 팀을 함께 조회해서 지연 로딩 발생 안 함  
 System.out.println("username = " + member.getUsername() + ", " + "teamname = " + member.getTeam().getName());  
}

🡺회원 엔티티를 조회하면 연관된 팀 엔티티도 함께 조회한다.

🡺m.team 다음에 별칭이 없는데 페치 조인은 별칭을 사용할 수 없다.

-주의할 점은 페치는 지연로딩 발생하지 않기 때문에 프록시가 아닌 실제 엔티티다.

**컬렉션 페치 조인**

String jpql = "select t from Team t join fetch t.members where t.name = 'team1'";  
List<Team> teams = em.createQuery(jpql, Team.class).getResultList();  
  
for (Team team : teams) {  
 System.*out*.println("teamname = " + team.getName() + ", team = " + team);  
  
 for (Member member : team.getMembers()) {  
 //페치 조인으로 팀과 회원을 함께 조회해서 지연 로딩 발생 안 함  
 System.*out*.println("->username = " + member.getUsername() + ", member = " + member);  
 }  
}

🡺팀 엔티티를 조회 하면 회원 엔티티로 함께 조회한다.

**참고**

-일대다 조인은 결과가 증가할 수 있지만 일대일, 다대일 조인은 결과가 증가하지 않는다.

**페치 조인과 DISTINCT**

-JPQL의 DISTINCT 명령어는 SQL에 DISTINCT를 추가하는 것은 물론이고 애플리케이션에서 한 번 더 중복을 제거한다.

String jpql = "select distinct t from Team t join fetch t.members where t.name = 'team1'";  
// 원래 결과가 하나만 나와야 되는데 ...  
List<Team> teams = em.createQuery(jpql, Team.class).getResultList();  
  
for (Team team : teams) {  
 System.*out*.println("teamname = " + team.getName() + ", team = " + team);  
  
 for (Member member : team.getMembers()) {  
 //페치 조인으로 팀과 회원을 함께 조회해서 지연 로딩 발생 안 함  
 System.*out*.println("->username = " + member.getUsername() + ", member = " + member);  
 }  
}

**페치 조인과 일반 조인의 차이**

-JPQL은 결과를 반환할 때 연관관계까지 고려하지 않는다. 단지 SELECT 절에 지정한 엔티티만 조회할 뿐이다. (Hibernate에서는??)

**페치 조인의 특징과 한계**

-페치 조인을 사용하면 SQL 한 번으로 연관된 엔티티들을 함꼐 조회할 수 있어서 호출 횟수를 줄여준다.

-페치 조인은 글로벌 로딩 전략보다 우선한다. (ex @OneToMant(fetch = FetchType.LAZY)

-하지만 전체로 보면 사용하지 않는 엔티티를 자주 로딩하므로 오히려 성능에 영향을 미칠 수 있다.

**페치 조인의 한계**

-페치 조인 대상에는 별칭을 줄 수 없다.

-둘 이상의 컬렉션을 페키할 수 없다.

-컬렉션을 페치 조인하면 페이징 API를 사용할 수 없다.

**10.2.8 경로 표현식**

-쉽게 이야기해서 .(점)을 찍어 객체 그래프를 탐색하는 것이다.

**경로 표현식의 용어 정리**

-상태 필드(state field) : 단순히 값을 저장하기 위한 필드

-연관 필드(association field) : 연관관계를 위한 필드, 임베디드 타입 포함(필드 or 프로퍼티)

* 단일 값 연관 필드 : @ManyToOne, @OneToOne, 대상이 엔티티
* 컬렉션 값 연관 필드 : @OneToMany, @ManyToMany, 대상이 컬렉션

**경로 표현식과 특징**

-상태 필드 경로 : 경로 탐색의 끝이다. 더는 탐색할 수 없다.

-단일 값 연관 경로 : 묵시적으로 내부 조인이 일어난다. 단일 값 연관 경로는 계속 탐색할 수 있다.

-컬렉션 값 연관 경로 : 묵시적으로 내부 조인이 일어난다. 더는 탐색할 수 없다. 단 FROM 절에서 조인을 통해 별칭을 얻으면 별칭으로 탐색할 수 있다.

**-상태 필드 경로 탐색**

select m.username, m.age from Member m (JPQL)

🡺select m.name, m.age from Member m (SQL)

**-단일 값 연관 경로 탐색**

select o.member from Order o (JPQL)

🡺select m.\* from Orders o inner join Member m on o.member\_id = m.id (SQL) 묵시적 조인이 일어났다.

**-명시적 조인** : JOIN을 직접 적어주는 것

ex) SELECT m FROM Member m JOIN m.team t

**-묵시적 조인** : 경로 표현식에 의해 묵시적으로 조인이 일어나는 것, 내부 조인INNER JON만 할 수 있다.

ex) SELECT m.team FROM Member m

**-경로 탐색을 사용한 묵시적 조인 시 주의사항**

* 항상 내부조인이다.
* 컬렉션은 경로 탐색의 끝이다. 컬렉션에서 경로 탐색을 하려면 명시적으로 조인해서 별칭을 얻어야 한다.
* 경로 탐색은 주로 SELECT, WHERE 절에서 사용하지만 묵시적 조인으로 인해 SQL의 FROM 절에 영향을 준다.

결론 : 묵시적 조인보다는 명시적 조인을 사용하자 (알아보기 힘들다.)

**10.2.11 다형성 쿼리**

-JPQL로 부모 엔티티를 조회하면 그 자식 엔티티도 함께 조회한다.

**TYPE**

-TYPE은 엔티티의 상속 구조에서 조회 대상을 특정 자식 타입으로 한정할 때

ex) JPQL

select i from Item i where type(i) IN (Book, Movie)

**TREAT(JAP2.1)**

-자바의 타입 캐스팅과 비슷하다. 상속 구조에서 부모 타입을 특정 자식 타입으로 다룰 때 사용

**10.2.14 엔티티 직접 사용**

**기본 키 값**

Member member = em.find(Member.class, 1L);  
  
String qlString = "select m from Member m where m = :member";  
List resultList = em.createQuery(qlString)  
 .setParameter("member", member)  
 .getResultList();

**외래 키 값**

Team team = em.find(Team.class, 3L);  
  
String qlString = "select m from Member m where m.team = :team";  
List resultList = em.createQuery(qlString)  
 .setParameter("team", team)  
 .getResultList();

**10.2.15 Named 쿼리: 정적 쿼리**

-동적 쿼리 : em.createQuery(“select ..”) 처럼 JPQL을 문자로 완성해서 직접 넘기는 것

-정적 쿼리 : 미리 정의한 쿼리에 이름을 부여해서 필요할 때 사용할 수 있는데 이것을 Named 쿼리라고 한다. (변경 불가)

🡺성능상 이점이 있다.

**Named 쿼리를 어노테이션에 정의**

@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", "kim")  
 .getResultList();

🡺Named 쿼리를 사용할 때는 em.createNamedQuery() 메소드에 Naemd 쿼리 이름을 입력하면 된다.

여러 개일 경우

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

. .

}

**@NamedQuery**

@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface NamedQuery {  
 String name();  
  
 String query();  
  
 LockModeType lockMode() default LockModeType.NONE;  
  
 QueryHint[] hints() default {};  
}

-lockMode : 쿼리 실행 시 락을 건다.

-hints : SQL 힌트가 아닌 JPA 구현체에게 제공하는 힌트다.

**Named 쿼리를 XML에 정의**

-<![CDATA[ ]]>를 사용하면 그 사이에 문장을 그대로 출력하므로 예약 문자도 사용할 수 있다.

**참고**

META-INF/orm.xml 은 기본 매핑파일이기 때문에 별도 설정 하지 않아도 되지만

<persistence-unit name="jpabook">  
 <mapping-file>META-INF/ormMember.xml</mapping-file>

다음과 같이 특정 파일은 mapping-file 로 직접 경로 설정해 주어야 한다.

-XML과 어노테이션에 같은 설정이 있으면 XML이 우선권이 있다.

**10.3 Criteria**

-JPQL을 자바 코드로 작성하도록 도와 주는 빌더 클래스 API

-문자가 아닌 코드로 JPQL을 작성하므로 문법 오류를 컴파일 단계에서 잡을 수 있다.

-코드가 복잡하고 장황해서 직관적으로 이해하기가 힘들다.

**10.3.1 Criteria 기초**

CriteriaBuilder cb = em.getCriteriaBuilder(); // Criteria 빌더를 얻어야 한다.  
  
CriteriaQuery<Member> cq = cb.createQuery(Member.class); // 쿼리 빌더에서 Criteria 쿼리를 생성  
  
Root<Member> m = cq.from(Member.class); // FROM 절 생성. m은 별칭으로 조회의 시작점이다.  
cq.select(m); // SELECT 절  
  
TypedQuery<Member> query = em.createQuery(cq); // JPQL과 같다.  
List<Member> members = query.getResultList();

**검색 조건 추가**

CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Member> cq = cb.createQuery(Member.class);  
Root<Member> m = cq.from(Member.class);  
  
// 검색 조건 정의  
Predicate usernameEual = cb.equal(m.get("username"), "kim");  
  
// 정렬 조건 정의  
Order ageDesc = cb.desc(m.get("age"));  
  
// 쿼리 생성  
cq.select(m)  
 .where(usernameEual) // WHERE 절 생성  
 .orderBy(ageDesc);  
  
List<Member> members = em.createQuery(cq).getResultList();

**쿼리 루트와 별칭**

-Root<Member> m = …; 여기서 m이 쿼리 루트

-쿼리 루트는 조회의 시작점이다.

-Criteria에서 사용되는 특별한 별칭이다.

-별칭은 엔티티에만 부여 가능

**10.3.2 Criteria 쿼리 생성**

-CriteriaBuilder는 제네릭으로 되어 있어서 어떤 타입으로든 받을 수 있다.

public interface CriteriaBuilder {  
 CriteriaQuery<Object> createQuery();  
 <T> CriteriaQuery<T> createQuery(Class<T> var1);

CriteriaQuery<Member> cq = cb.createQuery(Member.class);

🡺반환 타입을 지정할 수 없을 때는 Object로

🡺반환 타입이 둘 이상이면 Object[]를 사용

**10.3.3 조회**

public interface CriteriaQuery<T> extends AbstractQuery<T> {  
 CriteriaQuery<T> select(Selection<? extends T> var1); // 한건 지정  
 CriteriaQuery<T> multiselect(Selection... var1); // 여러 건 지정  
 CriteriaQuery<T> multiselect(List<Selection<?>> var1); // 여러 건 지정

…

}

**DISTINCT**

CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);  
Root<Member> m = cq.from(Member.class);  
cq.multiselect(m.get("username"), m.get("age")).distinct(true);  
//cq.select(cb.array(m.get("username"), m.get("age"))).distinct(true);  
  
TypedQuery<Object[]> query = em.createQuery(cq);  
List<Object[]> resultList = query.getResultList();

**NEW, construct()**

-JPQL에서 select new 생성자() 구문을 Criteria에서는 츄.construct(클래스 타입, ..)로 사용한다.

CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<MemberDTO> cq = cb.createQuery(MemberDTO.class);  
Root<Member> m = cq.from(Member.class);  
  
cq.select(cb.construct(MemberDTO.class, m.get("username"), m.get("age")));  
  
TypedQuery<MemberDTO> query = em.createQuery(cq);  
List<MemberDTO> resultList = query.getResultList();

🡺JPQL에서는 select new jpabook.domain.MemeberDTO() 같이 패키지명을 다 적어 주었다.

**튜플**

-Criteria는 Map과 비슷한 튜플이라는 기능을 제공한다.

CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Tuple> cq = cb.createTupleQuery();  
// CriteriaQuery<Tuple> cq = cb.createQuery(Tuple.class);  
  
Root<Member> m = cq.from(Member.class);  
cq.multiselect(  
 m.get("username").alias("username"), // 튜플 별칭  
 m.get("age").alias("age")  
);  
  
TypedQuery<Tuple> query = em.createQuery(cq);  
List<Tuple> resultList = query.getResultList();  
for (Tuple tuple : resultList) {  
 // 튜플 별칭으로 조회  
 String username = tuple.get("username", String.class);  
 Integer age = tuple.get("age", Integer.class);  
}

🡺이름 기반이므로 순서 기반의 Object[] 보다 안전하다.

**10.3.4 집합**

**GROUP BY**

CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);  
Root<Member> m = cq.from(Member.class);  
  
Expression maxAge = cb.max(m.<Integer>get("age"));  
Expression minAge = cb.min(m.<Integer>get("age"));  
  
cq.multiselect(m.get("team").get("name"), maxAge, minAge);  
cq.groupBy(m.get("team").get("name")); // group by  
  
TypedQuery<Object[]> query = em.createQuery(cq);  
List<Object[]> resultList = query.getResultList();

**HAVING**

cq.multiselect(m.get("team").get("name"), maxAge, minAge);  
cq.groupBy(m.get("team").get("name"));  
cq.having(cb.gt(minAge, 10));

🡺위 조건에서 having 절만 추가 했다.

**10.3.5 정렬**

// 정렬 조건 정의  
Order ageDesc = cb.desc(m.get("age"));  
  
// 쿼리 생성  
cq.select(m)  
 .where(usernameEual) // WHERE 절 생성  
 .orderBy(ageDesc);

**10.3.6 조인**

-조인은 join() 메소드와 JoinType 클래스를 사용한다.

CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);  
Root<Member> m = cq.from(Member.class);  
Join<Member, Team> t = m.join("team", JoinType.*INNER*.*INNER*); // 내부 조인  
  
cq.multiselect(m, t)  
 .where(cb.equal(t.get("name"), "team1"));  
  
TypedQuery<Object[]> query = em.createQuery(cq);  
List<Object[]> resultList = query.getResultList();

**10.3.7 서브 쿼리**

**간단한 서브 쿼리**

CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Member> mainQuery = cb.createQuery(Member.class);  
  
Subquery<Double> subquery = mainQuery.subquery(Double.class);  
  
// 서브 쿼리  
Root<Member> m2 = subquery.from(Member.class);  
subquery.select(cb.avg(m2.<Integer>get("age")));  
  
// 메인 쿼리  
Root<Member> m = mainQuery.from(Member.class);  
mainQuery.select(m)  
 .where(cb.ge(m.<Integer>get("age"), subquery));  
  
TypedQuery<Member> query = em.createQuery(mainQuery);  
List<Member> resultList = query.getResultList();

**상호 관련 서브 쿼리**

-correlate 메소드를 사용하면 메인 쿼리의 별칭을 서브 쿼리에서 사용할 수 있다.

**10.3.8 IN 식**

CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Member> cq = cb.createQuery(Member.class);  
Root<Member> m = cq.from(Member.class);  
  
cq.select(m)  
 .where(cb.in(m.get("username"))  
 .value("kim")  
 .value("sim"));  
  
List<Member> members = em.createQuery(cq).getResultList();

🡺CriteriaBuilder의 in 사용

**10.3.9 CASE 식**

-selectCase(), when(), otherwise() 메소드 사용

CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);  
  
Root<Member> m = cq.from(Member.class);  
cq.multiselect(  
 m.get("username"),  
 cb.selectCase()  
 .when(cb.ge(m.<Integer>get("age"), 60), 600)  
 .when(cb.le(m.<Integer>get("age"), 15), 500)  
 .otherwise(100)  
);  
  
TypedQuery<Object[]> query = em.createQuery(cq);  
List<Object[]> resultList = query.getResultList();

**10.3.10 파라미터 정의**

CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Member> cq = cb.createQuery(Member.class);  
  
Root<Member> m = cq.from(Member.class);  
  
cq.select(m)  
 .where(cb.equal(m.get("username"), cb.parameter(String.class, "usernameParma")));  
  
List<Member> resultList = em.createQuery(cq)  
 .setParameter("usernameParma", "kim")  
 .getResultList();

**10.3.11 네이티브 함수 호출**

CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Long> cq = cb.createQuery(Long.class);  
  
Root<Member> m = cq.from(Member.class);  
Expression<Long> function = cb.function("SUM", Long.class, m.get("age"));  
cq.select(function);

**10.3.12 동적 쿼리**

**JPQL**

// 검색 조건  
Integer age = 10;  
String username = null;  
String teamName = "팀A";  
  
//JPQL 동적 쿼리 생성  
StringBuilder jpql = new StringBuilder("select m from Member m join m.team t ");  
List<String> criteria = new ArrayList<String>();  
  
if (age != null) {  
 criteria.add(" m.age = :age ");  
}  
  
if (username != null) {  
 criteria.add(" m.username = :username ");  
}  
  
if (teamName != null) {  
 criteria.add(" t.name = :teamName ");  
}  
  
if (criteria.size() > 0) {  
 jpql.append(" where ");  
}  
  
for (int i = 0; i < criteria.size(); i++) {  
 if (i > 0) {  
 jpql.append(" and ");  
 }  
 jpql.append(criteria.get(i));  
}  
  
TypedQuery<Member> query = em.createQuery(jpql.toString(), Member.class);  
  
if (age != null) {  
 query.setParameter("age", age);  
}  
  
if (username != null) {  
 query.setParameter("username", username);  
}  
  
if (teamName != null) {  
 query.setParameter("teamName", teamName);  
}  
  
List<Member> resultList = query.getResultList();

🡺컴파일 시점에 알기 어렵다.. 나도 select 오타 내서 계속 에러 났다 …

**Criteria**

// 검색 조건  
Integer age = 10;  
String username = null;  
String teamName = "팀A";  
  
// Criteria 동적 쿼리 생성  
CriteriaBuilder cb = em.getCriteriaBuilder();  
CriteriaQuery<Member> cq = cb.createQuery(Member.class);  
  
Root<Member> m = cq.from(Member.class);  
Join<Member, Team> t = m.join("team");  
  
List<Predicate> criteria = new ArrayList<Predicate>();  
  
if (age != null) {  
 criteria.add(cb.equal(m.<Integer>get("age"), cb.parameter(Integer.class, "age")));  
}  
  
if (username != null) {  
 criteria.add(cb.equal(m.get("username"), cb.parameter(String.class, "username")));  
}  
  
if (teamName != null) {  
 criteria.add(cb.equal(t.get("name"), cb.parameter(String.class, "teamName")));  
}  
  
cq.where(cb.and(criteria.toArray(new Predicate[0])));  
  
TypedQuery<Member> query = em.createQuery(cq);  
  
if (age != null) {  
 query.setParameter("age", age);  
}  
  
if (username != null) {  
 query.setParameter("username", username);  
}  
  
if (teamName != null) {  
 query.setParameter("teamName", teamName);  
}  
  
List<Member> resultList = query.getResultList();

🡺공백이나 문법 오류 때문에 에러가 발생하지는 않지만 여전히 복잡함….

**10.3.13 함수 정리**

책 참조

**10.4 QueryDSL**

-Criteria가 너무 복잡해서 생긴 것이 QueryDSL

**10.4.1 QueryDSL 설정**

**pom.xml**

**필요 라이브러리**

<dependency>  
 <groupId>com.querydsl</groupId>  
 <artifactId>querydsl-apt</artifactId>  
 <version>${querydsl.version}</version>  
</dependency>  
<dependency>  
 <groupId>com.querydsl</groupId>  
 <artifactId>querydsl-jpa</artifactId>  
 <version>${querydsl.version}</version>  
</dependency>

-querydsl-jpa : QueryDSL JPA 라이브러리

-querydsl-api : 쿼리 타입(Q)을 생성할 때 필요한 라이브러리

**환경 설정**

<build>  
 <plugins>  
 <plugin>  
 <groupId>com.mysema.maven</groupId>  
 <artifactId>apt-maven-plugin</artifactId>  
 <version>1.1.3</version>  
 <executions>  
 <execution>  
 <goals>  
 <goal>process</goal>  
 </goals>  
 <configuration>  
 <outputDirectory>target/generated-sources/java</outputDirectory>  
 <processor>com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

-QueryDSL을 사용하려면 Criteria의 메타 모델처럼 쿼리용 클래스를 생성해야 된다.

**10.4.2 시작**

//조회  
JPAQueryFactory query = new JPAQueryFactory(em);  
QMember qMember = QMember.*member*;  
  
//쿼리, 결과 조회  
List<Member> members =  
 query.selectFrom(qMember)  
 .where(qMember.username.eq("kim"))  
 .orderBy(qMember.username.desc())  
 .fetch();

4.x 대여서 JPAQuery를 사용하지 않고 JPAQueryFactory를 사용했다.

**10.4.3 검색 조건 쿼리**

//조회  
JPAQueryFactory query = new JPAQueryFactory(em);  
QMember qMember = QMember.*member*;  
  
//쿼리, 결과 조회  
List<Member> members =  
 query.selectFrom(qMember)  
 .where(qMember.username.eq("kim").and(qMember.age.gt(10)))  
 .fetch();

-QueryDSL의 where절 에는 and나 or를 사용할 수 있다.

**10.4.4 결과 조회**

-uniqueResult : 조회 결과가 한 건일 때 사용한다.

-singleresult : uniqueResult와 같지만 결과가 하나 이상이면 처음 데이터를 반환

-list : 결과가 하나 이상일 때 사용

**10.4.5 페이징과 정렬**

List<Member> members =  
 query.selectFrom(qMember)  
 //.where(qMember.username.eq("kim").and(qMember.age.gt(10)))  
 .where((qMember.age.gt(10)))  
 .orderBy(qMember.age.desc())  
 .offset(10).limit(20)  
 .fetch();

-정렬은 asc(), desc()를 사용

-페이징은 offset과 limit을 적절히 조합해서 사용

QueryModifiers queryModifiers = new QueryModifiers(10L, 20L); // limit, offset  
  
List<Member> members2 =  
 query.selectFrom(qMember)  
 //.where(qMember.username.eq("kim").and(qMember.age.gt(10)))  
 .where((qMember.age.gt(10)))  
 .restrict(queryModifiers)  
 .fetch();

-페이징은 restrict를 사용할 수 있다.

// 실제 페이징 처리  
QueryResults result = query.from(qMember)  
 .where(qMember.age.gt(10))  
 .offset(0).limit(10)  
 .fetchResults();  
  
long total = result.getTotal(); // 검색된 전체 데이터 수  
long limit = result.getLimit();  
long offse = result.getOffset();  
List<Member> results = result.getResults(); // 조회된 데이터

-실제 페이징 할 때 total을 알아야 되는데 QueryResults를 사용하면 count를 한번 더 수행한다.

**10.4.6 그룹**

List<Member> members =  
 query.selectFrom(qMember)  
 .groupBy(qMember.age)  
 .having(qMember.age.gt(5))  
 .fetch();

**10.4.7 조인**

-조인은 innerJoin(join), leftJoin, rightJoin, fullJoin을 사용할 수 있고 추가로 JPQL의 on과 성능 최적화를 위한 fetch 조인도 사용할 수 있다.

-조인의 기본 문법은 첫 번째 파라미터에 조인 대상을 지정하고, 두 번째 파라미터에 별칭으로 사용할 쿼리 타입을 지정하면 된다.

List<Member> results = query.selectFrom(qMember)  
 .join(qMember.team, qTeam)  
 .fetch();

**10.4.8 서브 쿼리**

-JPASubQuery를 생성해서 사용한다. (상위버전에서는 JPAExpressions 사용한다.)

List<Member> results = query.selectFrom(qMember)  
 .where(qMember.age.eq(  
 JPAExpressions.*select*(qMemberSub.age.max()).from(qMemberSub)))  
 .fetch();

**10.4.9 프로젝션 결과 반환**

-select 절제 조회 대상을 지정하는 것을 프로젝션이라고 한다.

**빈 생성**

-Projections.bean() 수정자 setter를 사용해서 값을 채운다.

-Projections.fields() 는 필드에 직접 접근해서 값을 채워준다. (private로 설정해도 동작)

-Projections.constructor() 는 생성자를 사용한다. 물론 파라미터 순서가 같아야 한다.

**10.4.10 수정, 삭제 배치 쿼리**

**수정 (전 버전은 JPAUpdateClause)**

long count = query.update(qMember)  
 .where(qMember.age.gt(10))  
 .set(qMember.age, 40)  
 .execute();

**삭제 (전 버전은 JPADeleteCaluse)**

long count = query.delete(qMember)  
 .where(qMember.age.gt(20))  
 .execute();

**10.4.11 동적 쿼리**

SearchParam param = new SearchParam();  
param.setName("kim");  
param.setAge(20);  
  
JPAQueryFactory query = new JPAQueryFactory(em);  
QMember qMember = QMember.*member*;  
  
BooleanBuilder builder = new BooleanBuilder();  
  
if (param.getName() != null) {  
 builder.and(qMember.username.eq(param.getName()));  
}  
  
if (param.getAge() != null) {  
 builder.and(qMember.age.gt(param.getAge()));  
}  
  
List<Member> result = query.selectFrom(qMember)  
 .where(builder)  
 .fetch();

🡺BooleanBuilder 사용

**10.4.12 메소드 위임**

-메소드 위임 기능을 사용하면 쿼리 타입에 검색 조건을 직접 정의할 수 있다.

public class ItemExpression {  
 @QueryDelegate(Member.class)  
 public static BooleanExpression isGraterThen(QMember member, Integer age) {  
 return member.age.gt(age);  
 }  
}

🡺QueryDelegate 어노테이션 사용

@Generated("com.querydsl.codegen.EntitySerializer")  
public class QMember extends EntityPathBase<Member> {  
 public BooleanExpression isGraterThen(Integer age) {  
 return ItemExpression.*isGraterThen*(this, age);  
 }

}

🡺QMember안에 해당 메서드가 삽입 되었다.

JPAQueryFactory query = new JPAQueryFactory(em);  
QMember qMember = QMember.*member*;  
  
List<Member> result = query.selectFrom(qMember)  
 .where(qMember.isGraterThen(10))  
 .fetch();

**10.5 네이티브 SQL**

-JPQL은 표준 SQL이 지원하는 대부분 문법과 SQL 함수들을 지원하지만 특정 데이터베이스에 종속적인 기능은 지원하지 않는다.

* 특정 데이터베이스 종속 함수, 문법
* 인라인 뷰, UNION, INTERSECT
* 스토어드 프로시저

**특정 데이터베이스만 사용하는 함수**

-JPQL에서 네이티브 SQL 함수를 호출

-하이버네이트는 데이터베이스 방언에 대해 종속적인 함수들을 정의해 둠. 또는 직접 호출할 함수 정의

**특정 데이터베이스만 지원하는 SQL 쿼리 힌트**

-하이버네이트를 포함한 몇몇 JPA 구현체들이 지원

**인라인 뷰, UNIUON, INTERSECT**

-일부 JPA 구현체들 지원 (하이버네이트 미지원)

**스토어 프로시저**

-JQPL에서 스토어드 프로시저 호출 가능

**특정 데이터베이스만 지원하는 문법**

-네이티브 SQL 사용

**JPA 네이티브 SQL과 JDBC API 차이**

🡺엔티티 형태로 조회 할 수 있고 JPA가 지원하는 영속성 컨텍스트 사용 가능

**10.5.1 네이티브 SQL 사용**

**엔티티 조회**

String sql = "SELECT ID, AGE, NAME, TEAM\_ID " +  
 "FROM MEMBER WHERE AGE > ?";  
  
Query nativeQuery = em.createNativeQuery(sql, Member.class)  
 .setParameter(1, 20);  
  
List<Member> resultList = nativeQuery.getResultList();

🡺SQL만 직접 사용했을 뿐이지 나머지 JPQL이랑 처리 하는 방식은 같다.

🡺위치 기반 파라미터만 지원한다… (하지만 하이버네이트는 이름 기반 가능)

**값 조회**

String sql = "SELECT ID, AGE, NAME, TEAM\_ID " +  
 "FROM MEMBER WHERE AGE > ?";  
  
Query nativeQuery = em.createNativeQuery(sql) // 두번째 파라미터 사용하지 않으면 값 조회 이다.  
 .setParameter(1, 10);  
  
List<Object[]> resultList = nativeQuery.getResultList();  
for (Object[] row : resultList) {  
 System.*out*.println("id = " + row[0]);  
 System.*out*.println("age = " + row[1]);  
 System.*out*.println("name = " + row[2]);  
 System.*out*.println("team\_id = " + row[3]);  
}

**결과 매핑 사용**

-엔티티와 스칼라 값을 함께 조회하는 것처럼 매핑이 복잡해지면 @SqlResultSetMapping을 정의

**10.5.2 Named 네이티브 SQL**

-JPQL처럼 네이티브 SQL도 Named 네이티브 SQL 가능

@NamedNativeQuery(  
 name = "Member.memberSQL",  
 query = "SELECT ID, AGE ,NAME, TEAM\_ID " +  
 "FROM MEMBER WHERE AGE > ?",  
 resultClass = Member.class  
)  
public class Member {

...

}

TypedQuery<Member> nativeQuery = em.createNamedQuery("Member.memberSQL", Member.class)  
 .setParameter(1, 20);

**resultSetMapping**

@Entity  
// resultMapping  
@SqlResultSetMapping(name = "memberWithOrderCount",  
 entities = { @EntityResult(entityClass = Member.class) },  
 columns = { @ColumnResult(name = "ORDER\_COUNT") })  
// namedSQL  
@NamedNativeQuery(  
 name = "Member.memberSQL",  
 query = "SELECT ID, AGE ,NAME, TEAM\_ID " +  
 "FROM MEMBER WHERE AGE > ?",  
 resultSetMapping = "memberWithOrderCount"  
)

🡺resultSetMapping 사용 시, SqlResultSetMapping 같이 사용 가능

**10.5.3 네이티브 SQL XML에 정의**

**XML**

<named-native-query name="Member.memberWithOrderCountByXml" result-set-mapping="memberWithOrderCountResultMap">  
 <query>  
 <![CDATA[  
 SELECT M.ID, AGE, NAME, TEAM\_ID, I.ORDER\_COUNT  
 FROM MEMBER M  
 LEFT JOIN  
 (SELECT IM.ID, COUNT(\*) AS ORDER\_COUNT  
 FROM ORDERS O, MEMBER IM  
 WHERE O.MEMBER\_ID = IM.ID) I  
 ON M.ID = I.ID  
 ]]>  
 </query>  
</named-native-query>  
<sql-result-set-mapping name="memberWithOrderCountResultMap">  
 <entity-result entity-class="chapter10.nativequery.Member"/>  
 <column-result name="ORDER\_COUNT"/>  
</sql-result-set-mapping>

**java**

List<Object[]> resultList = em.createNamedQuery("Member.memberWithOrderCountByXml")  
 .getResultList();  
for (Object[] row : resultList) {  
 Member member = (Member) row[0];  
 BigInteger orderCount = (BigInteger) row[1];  
  
 System.*out*.println("member = " + member);  
 System.*out*.println("orderCount = " + orderCount);  
}

**10.5.4 네이티브 SQL 정리**

-네이티브 SQL은 JPQL이 자동 생성하는 SQL을 수동으로 직접 정의하는 것이다. 따라서 JPA가 제공하는 기능 대부분을 그대로 사용할 수 있다.

-네이티브 SQL은 JPQL 등을 이용해 보고 정 안되면 선택하자(차선책)

-네이티브로도 부족함을 느낀다면?? SQL 매퍼(MyBatis 같은) 사용

**10.5.5 스토어드 프로시저 (2.1 부터 지원)**

private static void storedProcedure(EntityManager em) {  
 StoredProcedureQuery spq = em.createStoredProcedureQuery("proc\_multiply");  
 spq.registerStoredProcedureParameter(1, Integer.class, ParameterMode.*IN*);  
 spq.registerStoredProcedureParameter(2, Integer.class, ParameterMode.*OUT*);  
  
 spq.setParameter(1, 100);  
 spq.execute();  
  
 Integer out = (Integer) spq.getOutputParameterValue(2);  
 System.*out*.println("out = " + out);  
}

🡺createStoredProcedureQuery() 사용

private static void sotredProcedureByParameter(EntityManager em) {  
 StoredProcedureQuery spq = em.createStoredProcedureQuery("proc\_multiply");  
 spq.registerStoredProcedureParameter("inParam", Integer.class, ParameterMode.*IN*);  
 spq.registerStoredProcedureParameter("outParam", Integer.class, ParameterMode.*OUT*);  
  
 spq.setParameter("inParam", 100);  
 spq.execute();  
  
 Integer out = (Integer) spq.getOutputParameterValue("outParam");  
 System.*out*.println("out = " + out);  
}

🡺파라미터 방식

**10.6 객체지향 쿼리 심화**

**10.6.1 벌크 연산**

-수백개 이상의 엔티티를 하나씩 처리하기에 시간이 오래 걸린다. 이럴 때 벌크 연산 사용.

**UPDATE 벌크 연산**

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();

🡺executeUpdate를 사용한다.

**DELETE 벌크 연산**

String qlString =  
 "delete from Product p " +  
 "where p.price < :price";  
  
int resultCount = em.createQuery(qlString)  
 .setParameter("stockAmount", 10)  
 .executeUpdate();

🡺동일하게 executeUpdate를 사용한다.

-하이버네이트는 INSERT 벌크 연산도 지원한다.

**벌크 연산의 주의점**

-영속성 컨텍스트를 무시하고 데이터베이스에 직접 쿼리한다.

// 상품 조회 가격은 1000  
Product productA =  
 em.createQuery("select p from Product p where p.name = :name", Product.class)  
 .setParameter("name", "productA")  
 .getSingleResult();  
  
// 출력 결과 1000  
System.*out*.println("productA 수정 전 = " + productA.getPrice());  
  
// 벌크 연산 수행으로 모든 상품 가격 10% 상승  
em.createQuery("update Product p set p.price = p.price \* 1.1")  
 .executeUpdate();  
  
// 출력 결과: 1000  
System.*out*.println("productA 수정 후 = " + productA.getPrice());

🡺영속성 컨텍스트를 통하지 않고 데이터베이스에 직접 쿼리 했기 때문에 전/후 결과 값이 같다.

**해결법**

1) em.refersh() 사용

2) 벌크 연산 먼저 실행

3) 벌크 연산 수행 후 영속성 컨텍스트 초기화

**10.6.2 영속성 컨텍스트와 JPQL**

**쿼리 후 영속 상태인 것과 아닌 것**

-JPQL로 엔티티를 조회하면 영속성 컨텍스트에서 관리되지만 엔티티가 아니면 영속성 컨텍스트에서 관리되지 않는다.

🡺물론 엔티티를 조회하면 해당 엔티티가 가지고 있는 임베디드 타입은 함께 수정됨

**JPQL로 조회한 엔티티와 영속성 컨텍스트**

-JPQL로 데이터베이스에서 조회한 엔티티가 영속성 컨텍스트에 이미 있으면 JPQL로 데이터베이스에서 조회하는 결과를 버리고 대신에 영속성 컨텍스트에 있던 엔티티를 반환한다. (식별자 값으로 비교)

-JPQL로 조회한 엔티티는 영속 상태다.

-영속성 컨텍스트에 이미 존재하는 엔티티가 있으면 기존 엔티티를 반환한다.

🡺왜 이렇게 할까? 기존에 수정 중인 데이터가 있으면 사라질 수도 있기 때문에

**find() vs JPQL**

-em.find() 메소드는 영속성 컨텍스트에서 먼저 찾고 없으면 데이터베이스에서 찾는다.

**그렇다면 JPQL은?**

-항상 데이터베이스에 SQL을 실행해서 결과를 조회한다.

**JPQL의 특징을 정리해보자**

-JPQL은 항상 데이터베이스를 조회

-JPQL로 조회한 엔티티는 영속 상태

영속성 컨텍스트에 이미 존재하면 기존 엔티티 반환

**10.6.3 JPQL과 플러시 모드**

-플러시는 영속성 컨텍스트의 변경 내역을 데이터베이스에 동기화하는 것이다.

-플러시가 일어나면 영속성 컨텍스트에 등록, 수정, 삭제한 엔티티를 찾아서 INSERT, UPDATE, DELETE 수행을 한다.

**쿼리와 플러시 모드**

Product product =  
 em.createQuery("select p from Product p where p.name = :name", Product.class)  
 .setParameter("name", "productA")  
 .getSingleResult();  
  
em.setFlushMode(FlushModeType.*COMMIT*); // 커밋 시에만 플러시  
  
product.setPrice(2000L);  
  
// 가격이 2000원인 상품 조회  
Product product2 =  
 em.createQuery("select p from Product p where p.price = 2000", Product.class)  
 .getSingleResult();

🡺COMMIT 모드로 하면 플러시가 수행되지 않았기 때문에 쿼리 에러가 난다.



**해결 방법**

// 상품 조회 가격은 1000  
Product product =  
 em.createQuery("select p from Product p where p.name = :name", Product.class)  
 .setParameter("name", "productA")  
 .getSingleResult();  
  
em.setFlushMode(FlushModeType.*COMMIT*); // 커밋 시에만 플러시  
  
product.setPrice(2000L);  
  
// 해결 방법  
//em.flush(); // 1. flush 직접 호출  
  
// 가격이 2000원인 상품 조회  
Product product2 =  
 em.createQuery("select p from Product p where p.price = 2000", Product.class)  
 .setFlushMode(FlushModeType.*AUTO*) // 2. setFlushMode() 설정  
 .getSingleResult();

🡺flush를 직접 호출 하거나, setFlushMode를 설정하거나

**왜 commit을 사용하는 것일까?**

-쿼리시 발생하는 플러시 횟수를 줄여서 성능을 최적화

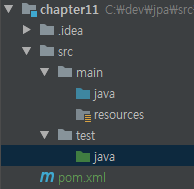
**CHAPTER11**

**11.1 프로젝트 환경설정**

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.tomcat.maven</groupId>  
 <artifactId>tomcat7-maven-plugin</artifactId>  
 <version>2.2</version>  
 <configuration>  
 <path>/</path>  
 <uriEncoding>UTF-8</uriEncoding>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

🡺톰켓 플러그인 사용

**11.1.1 프로젝트 구조**



**11.1.2 메이븐과 사용 라이브러리 관리**

-라이브러리 간에 충돌을 피하려면 groupId + artifactId는 유일해야 한다.

-핵심 라이브러리

* 스프링 MVC(spring-webmvc) : 스프링 mvc 라이브러리
* 스프링 ORM(spring-orm) : 스프링 프레임워크와 JPA를 연동하기 위한 라이브러리
* JPA, 하이버네이트(hibernate-entitiymanager) : JPA 표준과 하이버네이트를 포함하는 라이브러리

🡺hibernate-core, hibernate-jpa (연관 라이브러리)

-기타 라이브러리

* H2 데이터베이스 : 임베디드 데이터베이스로, JVM 메모리 안에서 동작 기능도 있음.
* 커넥션 풀(tomcat-jdbc) : jdbc 커넥션풀 사용
* WEB : 서블릿, JSP와 관련된 라이브러리
* 로깅 SLF4J & LogBack : 로깅 관련 라이브러리
* 테스트(spring-test) : 테스트용 라이브러리

-메이븐 <dependency>의 <scope> 설정

* compile(기본값) : 컴파일 시 라이브러리를 사용
* runtime : 실행 시 라이브러리를 사용
* provided : 외부에서 라이브러리가 제공. 컴파일 시 사용하지만 빌드에 포함하지 않는다.
* test : 테스트 코드에만 사용