JPA & Hibernate

JPA & Hibernate

1.1 ORM (Object-Relation Mapping) 이란?

“ 객체(Object) 와 관계형 데이터베이스(RDBMS)를 매핑(Mapping) 한다. ”

1.1.1 관계형 데이터베이스(RDBMS)를 저장소(Repository)로 사용하는 자바(Java) 애플리케이션

1) JDBC API를 직접 사용하는 코딩

2) MyBatis 나 Spring의 JdbcTemplate 같은 SQL Mapper를 사용하는 코딩

3) CRUD용 SQL를 계속 반복적으로 사용해야 한다. [SQL의 문제점]

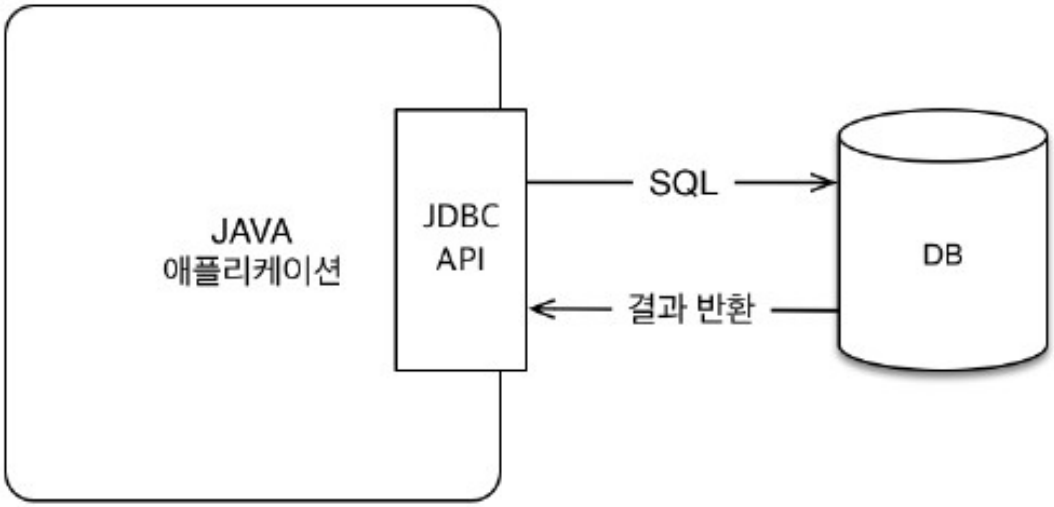
4) CRUD용 SQL를 자동 생성하는 도구들을 사용해도 애플리케이션 완성 단계에서 발생하는 추가 요구사항 해결에 많은 문제점이 발생한다. [SQL의 문제점]

5) 객체 모델링 보다는 데이터 중심 모델링(테이블 설계)를 우선시하고 중요하게 다룬다.[패러다임의 불일치]

6) 객체 지향의 장점(추상화, 캡슐화, 정보은닉, 상속, 다형성)을 포기, 단순히 객체는 데이터 전달 목적(DTO패턴) 에만 사용한다.[패러다임의 불일치]

7) 객체 지향 모델링을 정밀하게 할 수록 객체를 데이터베이스에서 저장하고 조회하는 작업이 어려워 진다. [패러다임의 불일치]

1.1.2 SQL의 문제점



JDBC API & SQL

1) 반복 작업

- 객체를 CRUD에 저장하려면 많은 SQL문과 JDBC API를 코드로 작성해야 한다.
- 테이블 마다 비슷한 작업을 하는데, 테이블 수 만큼 비슷한 작업을 해 주어야 한다.
- DAO 개발은 반복되는 작업으로 지루함과 반복의 연속이다.

1.1 ORM (Object-Relation Mapping) 이란?

2) SQL에 의존적인 개발

- Entity 객체에 필드 하나만 추가해도 많은 SQL문을 수정해야 하고 반대도 마찬가지다.

- 데이터 접근(Repository)계층에 SQL를 숨겨도 DAO를 열어 SQL문을 확인해야 한다.

- 물리적으로 SQL과 JDBC API를 데이터 접근 계층에 숨기고 분리하여도 Entity 객체에 강한 의존성을 가지고 있다. 반대도 마찬가지이다.

- 진정한 의미의 계층 분할이 어렵다.

- Entity를 신뢰할 수 없으며 SQL 의존적 개발 역시 피할 수 없다.

1.1.3 패러다임의 불일치

1) 객체 지향 프로그래밍(Object Oriented Programming)

- 애플리케이션이 발전하면 내부 복잡성(Complexity)이 증가하게 되며 복잡성을 제어할 수 있어야 지속적인 애플리케이션 개발이 가능하며 유지보수도 가능해 진다.

- OOP에서는 추상화, 캡슐화, 정보은닉, 상속, 다형성 등을 사용하여 애플리케이션(또는 시스템)의 복잡성을 제거 또는 제어할 수 있기 때문에 현대의 복잡한 애플리케이션 대부분은 객체 지향 언어로 개발하게 된다.

- 비지니스 요구사항을 도메인 모델을 객체로 모델링 하게 되면 객체 지향의 장점들을 활용할 수 있다.

2) 관계형 데이터베이스(Relational Database)

- 데이터 중심(테이블)으로 구조화에 목적을 두고 있다.

- 일련의 정형화된 테이블로 구성된 데이터 항목들의 집합체로 집합적 사고방식으로 데이터 저장/검색을 하게 된다.

- 따라서 당연히 추상화, 캡슐화, 정보은닉, 상속, 다형성과 같은 OOP 개념은 없다.

- 저장 검색을 위해서는 SQL이라는 질의언어를 사용해야 한다.

1.1 ORM (Object-Relation Mapping) 이란?

3) 패러다임의 불일치

- 객체 지향 프로그래밍으로 작성된 애플리케이션의 객체를 영구(persistence) 저장할 때 발생한다.
- 단순한 객체는 객체 인스턴스의 상태인 속성만 저장했다가 필요할 때 불러와서 복구하면 된다.
- 하지만, 상속을 받은 객체, **다른 객체를 참조**하는 경우 그 객체의 상태를 저장하는 것은 쉽지 않다.
- 자바에서는 객체를 파일로 저장하는 객체 직렬화(Serialization) 기능을 자체 제공하고 있지만, 객체 검색이 어렵다.
- 객체의 저장을 위해 주로 관계형 데이터베이스를 사용하게 되는데 이 때, 객체와 관계형 데이터베이스는 지향하는 목적이 서로 다르기 때문에 둘의 기능과 표현 방법이 다르다 (패러다임의 불일치)
- 패러다임의 불일치는 객체 지향 언어로 개발된 애플리케이션 입장에서 보면, 객체 구조를 테이블 구조에 저장하는 데 한계와 제한을 가지게 된다.
- 지금 까지는 개발자가 중간에서 패러다임의 문제를 많은 반복되는 코드와 시간을 소비하면서 이 문제를 해결하여 왔다.

1.1.4 ORM(Object-Relational Mapping) 이란 무엇인가?

" 객체(Object) 와 관계형 데이터베이스(RDBMS)를 매핑(Mapping) 한다. "

1) 마치 자바 컬렉션에 저장하듯 ORM 프레임워크에 객체를 저장하면 ORM 프레임워크가 적절한 SQL문을 생성하고 데이터베이스에 저장해 준다.

2) ORM 프레임워크는 객체와 테이블을 매핑하여 패러다임의 문제를 개발자 대신 해결하여 준다. 개발자는 객체와 테이블의 매핑 방법만 ORM 프레임워크에 알려 주기만 하면 된다.

3) 개발자는 객체 측면에서는 정교한 객체 모델링을 할 수 있고 관계형 데이터베이스는 데이터베이스에 맞게 모델링 하면 된다.

4) 데이터 중심의 관계형 데이터베이스를 사용해도 개발자는 객체 지향 애플리케이션 개발에 집중할 수 있다.

5) 성숙한 객체 지향 언어에는 대부분 ORM 프레임워크가 있다.

6) 자바 진영에서도 다양한 ORM 프레임워크들이 있는데, 그 중에 **하이버네이트(Hibernate) 프레임워크**를 가장 많이 사용하며 패러다임의 불일치로 발생하는 거의 모든 문제를 해결해 주는 기술 성숙도가 높은 ORM 프레임워크 이다.

1.2 JPA(Java Persistence API) 소개

1.2.1 자바 ORM 기술 표준

1) 엔티티 빈(Entity Bean)

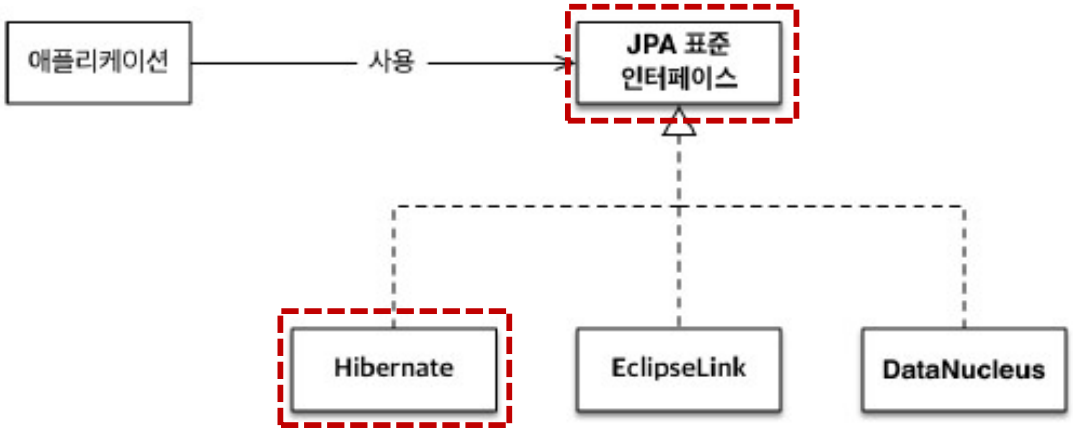
- 자바의 엔터프라이즈 자바 빈즈(EJB) 표준 기술에 포함된 ORM 기술이었다.
- 너무 복잡했으며 기술 성숙도도 떨어졌다.
- J2EE 애플리케이션에서만 동작하였다. (WAS가 필요)

2) 하이버네이트(Hibernate)

- 오픈 소스 ORM 프레임워크로 등장
- EJB ORM 기술보다 가볍고 실용적 이였으며 기술 성숙도가 높았다.
- 자바 엔터프라이즈 애플리케이션 서버 없이도 동작했기 때문에 많은 개발자들이 사용하기 시작했다.
- **EJB 3.0에서 하이버네이트를 기반으로 해서 새로운 ORM 기술 표준을 만들어 발표 하였는데, 이것이 JPA 이다.**

1.2 JPA(Java Persistence API) 소개

3) JPA(Java Persistence API)



- 자바 ORM 기술에 대한 API 표준 명세다.(현재 2.1, 2013) 즉, 인터페이스를 모아둔 것이다.
- JPA를 사용하기 위해서는 JPA를 구현한 ORM 프레임워크를 사용해야 한다.
- JPA 2.1를 구현한 ORM 프레임워크는 Hibernate, EclipseLink, DataNucleus 가 있는 데 이 중에 하이버네이트가 가장 대중적이다.
- JPA 표준때문에 특정 구현 기술에 대한 의존성을 주일 수 있고 다른 기술로의 이동도 가능하다.
- JPA는 ORM의 공통 기능들을 정의한 것으로 먼저 표준을 이해하고 특정 구현체의 고유 기능을 학습하는 것이 좋다.

1.2.2 JPA를 왜 사용해야 하는가?

1) 생산성

- JPA를 사용하면 자바 컬렉션에 객체를 저장하듯 개발자는 JPA에게 객체를 전달하면 된다.
- JDBC API를 사용하여 지루하고 반복적인 CRUD용 SQL를 직접 개발자가 작성하지 않아도 된다.
- 데이터베이스 설계 중심의 개발 패러다임을 객체 설계 중심으로 바꿀 수 있다.

2) 유지보수

- 엔티티 필드 하나의 수정은 DAO의 CRUD SQL문과 결과, 조건(파라미터)를 매핑하기 위한 JDBC API를 사용하는 자바 코드의 수정이 필요하다.
- JPA를 사용하게 되면 JPA가 이런 일들을 대신해주기 때문에 수정해야 할 코드가 많이 줄어든다.
- 즉, 개발자가 유지보수 해야 할 코드가 상당히 줄어들게 된다.
- JPA는 패러다임의 불일치 문제를 해결해 주기 때문에 유연하고 유지보수 하기 좋은 도메인 모델 중심의 객체지향 프로그래밍을 할 수 있다.

3) 패러다임 불일치 해결

객체가 가지고 있는 상속, 연관관계, 객체 그래프 탐색, 비교하기 등의 패러다임 불일치 문제를 해결하여 준다.

4) 성능

- 애플리케이션과 데이터베이스 사이에 최적화 기회를 제공한다.
- 자체에 객체를 저장하기 때문에 실제 저장되는 실제 데이터가 저장되는 데이터베이스 와의 사이에 캐시 기능을 제공한다.

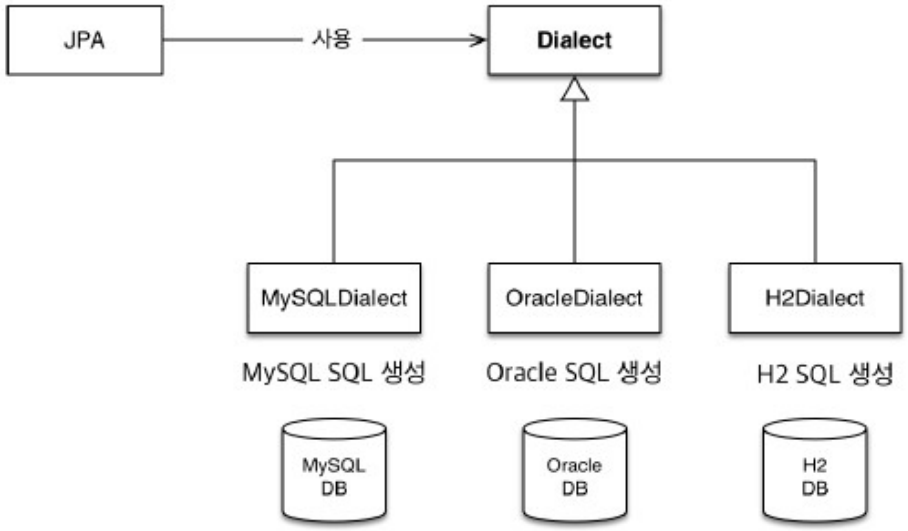
5) 표준

JPA는 자바 진영의 ORM 기술 표준으로 다른 기술로 손쉽게 변경이 가능하다.

6) 데이터 접근의 추상화 와 벤더 독립성

- 관계형 데이터베이스는 벤더마다 같은 기능도 사용법이 다른 경우가 있기 때문에 애플리케이션 개발 시 처음 선택한 데이터베이스에 종속되고 이 후, 다른 데이터베이스로 변경이 힘들다.

  예) 페이징, 키 값 자동 증가, 함수, VARCHAR, 등...



- JPA는 애플리케이션과 데이터베이스 사이에 추상화된 데이터 접근 계층을 제공해서 애플리케이션이 특정 데이터베이스에 종속되지 않게 한다.

- 데이터베이스의 변경은 JPA에게 설정을 통해 알려주기만 하면 된다.

1.2.3 ORM / JPA / Hibernate 에 대한 오해

1) ORM 프레임워크를 잘 쓰기 위해선 SQL문과 데이터베이스를 잘 몰라도 된다.

2) 성능이 느리다.

3) 통계 쿼리처럼 매우 복잡한 SQL문제의 해결이 가능한가?

4) MyBatis의 매핑 과의 ORM 매핑과의 차이점은?

6) 우리나라는 MyBatis를 많이 사용하는 데 하이버네이트 프레임워크 를 신뢰할 수 있는가?

[참고] http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014

7) 배우기가 매우 어렵다(학습 곡선이 높다)

JPA & Hibernate

2.1 데이터베이스

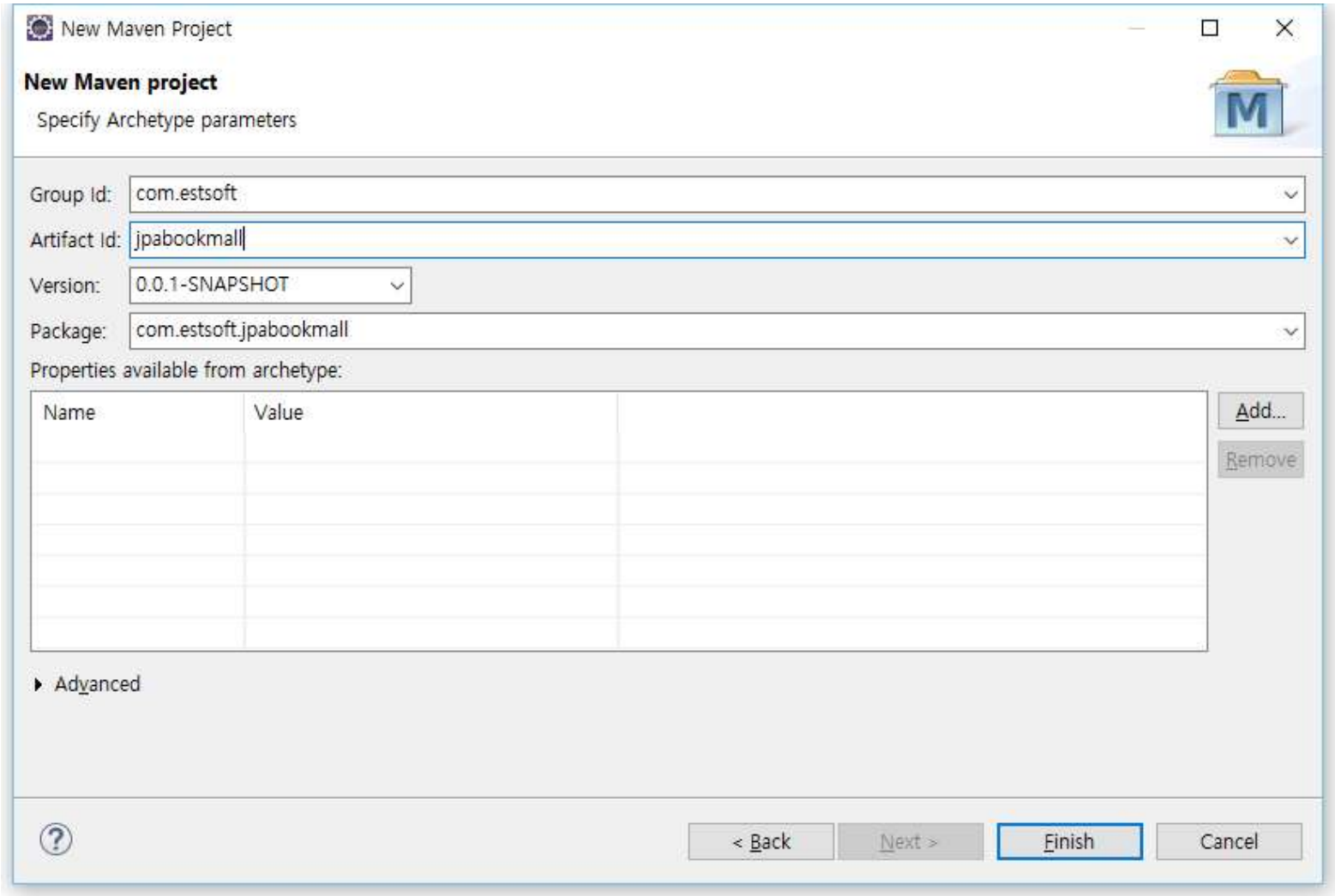
2.1.1 테스트 데이터베이스 생성(MySQL)

```
mysql > create database jpadb;
Query OK, 1row affected (0.08 sec)

mysql > grant all privileges on jpadb.* to 'jpadb'@'localhost' identified by 'jpadb';
Query OK, 1row affected (0.12 sec)

mysql >
```

2.1.2 테이블 생성(bookmall 도메인)

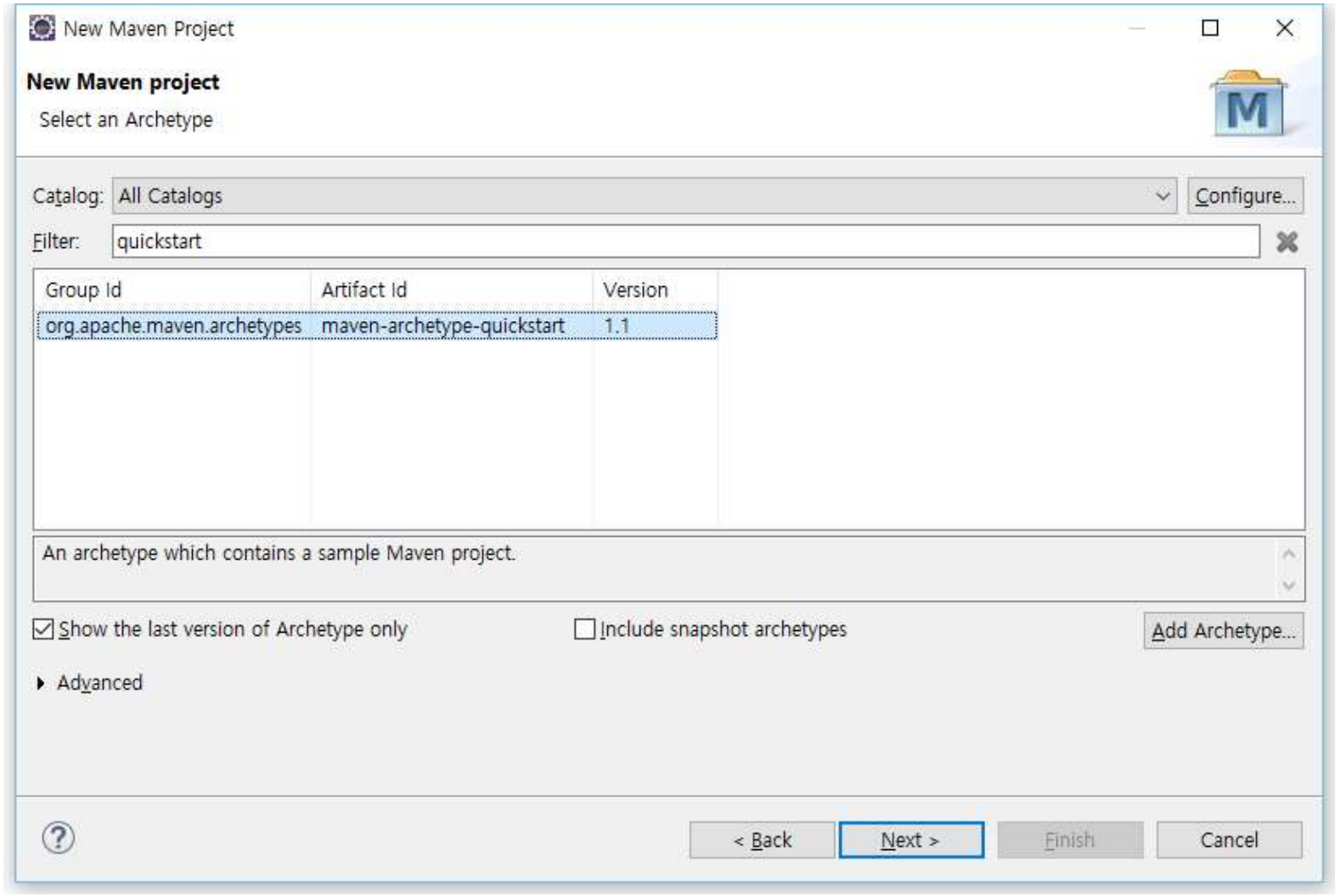
```
C:₩> mysql –D jpadb –u webdb -p
Enter Password: *****
Welcome to the MySQL monitor. Commands end with ; or ₩g
Your MySQL connection id is 3
Server version: 5.1.44-community MySQL Community Server (GPL)

Type 'help;' or '₩h' for help. Type '₩c' to clear the current input statement.

mysql > CREATE TABLE book (
    ->      no    INT UNSIGNED NOT NULL PRIMARY KEY,
    ->      title  VARCHAR(200) NOT NULL,
    ->      price MEDIUMINT     NOT NULL
    -> );
Query OK, 1row affected (0.12 sec)
```

2.2.2 프로젝트 생성


New Maven Project
New Maven project
Specify Archetype parameters
Group Id: com.estsoft
Artifact Id: jpabookmall
Version: 0.0.1-SNAPSHOT
Package: com.estsoft.jpabookmall
Properties available from archetype:
Name
Value
Add...
Remove
Advanced
< Back
Next >
Finish
Cancel

2.2 예제 프로젝트 jpabookmall

2.2.3 메이븐 프로젝트 (Artifact Id : maven-archetype-quickstart)



2.3 프로젝트 dependecy

2.3.1 JPA 구현체로 Hibernate를 사용하기 위한 핵심 라이브러리

1) hibernate-core
하이버네이트 코어 라이브러리

2) hibernate-entitymanager
하이버네이트가 JPA 구현체로 동작하도록 JPA 표준을 구현한 라이브러리

3) hibernate-jpa-2.1-api
JPA 2.1 표준 API를 모아둔 라이브러리

2.3 프로젝트 dependecy

2.3.2 pom.xml dependency 추가

1) hibernate-core. jar , hibernate-entitymanager.jar, hibernate-jpa-2.1-api.jar

```
<!-- JPA, Hibernate -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.0.12.Final</version>
</dependency>
```

2) MySQL JDBC Driver

```
<!-- MySQL JDBC Driver -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.38</version>
</dependency>
```

2.4 객체 매핑하기

2.4.1 book table

```
CREATE TABLE book (
    no    INT UNSIGNED NOT NULL PRIMARY KEY,   -- 번호 ( 기본키 )
    title  VARCHAR(200) NOT NULL,              -- 제목
    price MEDIUMINT    NOT NULL                -- 가격
 )
```

2.4.2 Book class [패키지 : com.estsoft.jpabookmall.domain]

```
public class Book {

    private Long no;                    // 번호
    private String title;               // 제목
    private Integer price;              // 가격

    // getter/setter
     public Long getNo() {
        return no;
     }
     public void setNo(Long no) {
        this.no = no;
     }
     public String getTitle() {
         return title;
     }
     public void setTitle(String title) {
         this.title = title;
     }
     public Integer getPrice() {
         return price;
     }
     public void setPrice(Integer price) {
         this.price = price;
     }
}
```

2.4 객체 매핑하기

2.4.3 클래스와 테이블 매핑

매핑 정보	객체	테이블
클래스와 테이블	Book	book
기본 키	no	no
필드와 컬럼	title	title
필드와 컬럼	price	price

JPA를 사용하기 위해서는 Book 클래스와 book 테이블을 매핑해야 한다. 다음과 같은 어노테이션을 클래스에 추가하여 매핑한다.

1) @Entity

클래스를 테이블과 매핑한다고 JPA에게 알려준다. 이렇게 @Entity 가 사용된 클래스를 **엔티티 클래스**라 한다.

2) @Table

엔티티 클래스에 매핑할 테이블 정보를 알려준다. name 속성을 사용해서 Book엔티티를 book 테이블에 매핑한다. 이 어노테이션을 생략하면 클래스 이름을 테이블 이름으로 자동매핑한다.

2.4 객체 매핑하기

3) @Id
엔티티 클래스의 필드를 테이블의 기본 키(Primary key)에 매핑 한다. 엔티티의 no 필드를 테이블의 no기본 키 컬럼에 매핑한다. 이렇게 @Id가 사용된 필드를 **식별자 필드**라 한다.

4) @Column
필드를 컬럼에 매핑한다. 여기서는 name 속성을 사용해서 엔티티의 title를 필드를 book 테이블의 title 컬럼에 매핑한다.

5) 매핑정보가 없는 필드
price 필드에는 매핑 어노테이션이 없다. 이렇게 매핑 어노테이션을 생략하면 필드명을 사용해서 컬럼명으로 매핑한다. 필드명이 price는 price 컬럼으로 자동 매핑된다.

주의
대소문자를 구분하는 데이터베이스를 사용하면 자동 매핑에 특히 주의해야 한다.

2.5 persistence.xml 설정

2.5.1 개요

1) JPA는 persistence.xml 을 사용해서 필요한 설정정보를 관리한다.

2) META-INF/pesistence.xml 클래스 패스 경로에 존재하면 별도의 설정 없이 JPA가 인식한다.

3) persistence.xml 설정 하기

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">

    <persistence-unit name="jpabookmall">

        <properties>

            <!-- 필수 속성 -->
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
            <property name="javax.persistence.jdbc.user" value="jpadb"/>
            <property name="javax.persistence.jdbc.password" value="jpadb"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/jpadb"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />

            <!-- 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>
```

2.5 persistence.xml 설정

2.5.2 persistence.xml 설정 내용 살펴보기

1) 영속성 유닛(Persistence Unit)

```
<persistence-unit name="jpabookmall">
```

- JPA 설정의 시작
- 일반적으로 연결할 데이터베이스당 하나의 영속성 유닛을 설정한다.
- 고유의 이름을 부여 하여야 한다.

2) JPA 표준 속성

javax.persistence로 시작하는 속성이며 JPA 표준 속성으로 특정 구현체에 종속되지 않는다.

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**

2.5 persistence.xml 설정

3) Hibernate 속성

hibernate 로 시작하는 속성은 하이버네이트 전용 속성이므로 하이버네이트에서만 사용할 수 있다.

hibernate.show_sql : 하이버네이트가 실행한 SQL을 출력한다.

hibernate.format_sql : 하이버네이트가 실행한 SQL을 출력할 때 보기 쉽게 정렬한다.

hibernate.use_sql_comments : 쿼리를 출력할 때 주석도 함께 출력한다.

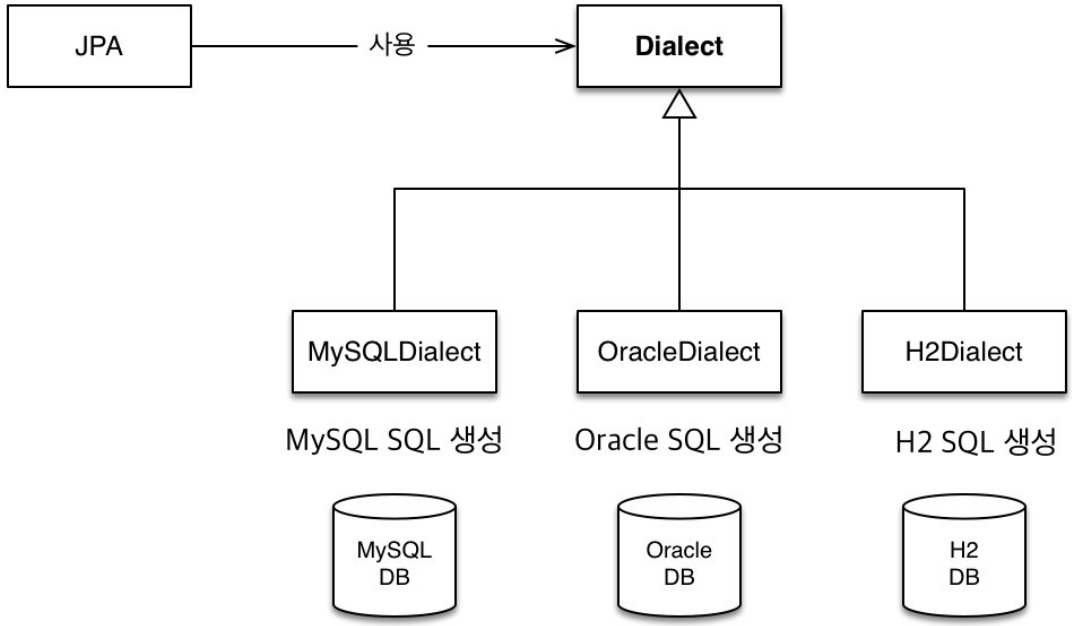
hibernate.id.new_generator_mappings : JPA 표준에 맞춘 새로운 키 생성 전략을 사용한다.

hibernate.dialect : 데이터베이스 방언

- JPA는 특정 데이터베이스에 종속적이지 않은 기술로 다른 데이터베이스로 손쉽게 교체할 수 있는 장점이 있다.
- 하지만, 각각의 데이터베이스가 제공하는 SQL 문법과 함수가 조금씩 다르다는 문제점이 있다.
- **SQL 표준을 지키지 않거나 특정 데이터베이스만의 고유한 기능을 JPA에서는 방언(Dialect)이라 한다.**
- 하이버네이트를 포함한 대부분의 JPA 구현체들은 다양한 데이터베이스 방언 클래스를 제공한다.
- 개발자는 JPA가 제공하는 표준 문법에 맞추어 JPA를 사용하면 되고, 특정 데이터베이스에 의존적인 SQL은 데이터베이스 방언이 처리해준다.

2.5 persistence.xml 설정

hibernate.dialect : 데이터베이스 방언



- 하이버네이트는 다양한 방언을 제공한다.

Oracle 10g : org.hibernate.dialect.Oracle10gDialect
MySQL : org.hibernate.dialect.MySQL5InnoDBDialect
DB2 : org.hibernate.dialect.DB2Dialect

참고 : http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single/#configuration-optional-dialects

2.6 애플리케이션 작성하기

2.6.1 전체 구조

```
// 1. 엔티티 매니저 팩토리 생성
    EntityManagerFactory emf = Persistence.createEntityManagerFactory( "jpabookmall" );

// 2. 엔티티 매니저 생성
    EntityManager em = emf.createEntityManager();

// 3. 트랜잭션 받기
    EntityTransaction tx = em.getTransaction();

// 4. 트랜젝션 시작
    tx.begin();

// 5. 비지니스 코드 실행
// logic( em );

// 6. 트랜잭션 커밋
    tx.commit();

// 7. 엔티티 매니저 종료
    em.close();

// 8. 엔티티 매니저 팩토리 닫기
    emf.close();
```

크게 3부분으로 나눌 수 있다.

1. 엔티티 매니저 설정
2. 트랜잭션 관리
3. 비즈니스 로직

2.6 애플리케이션 작성하기

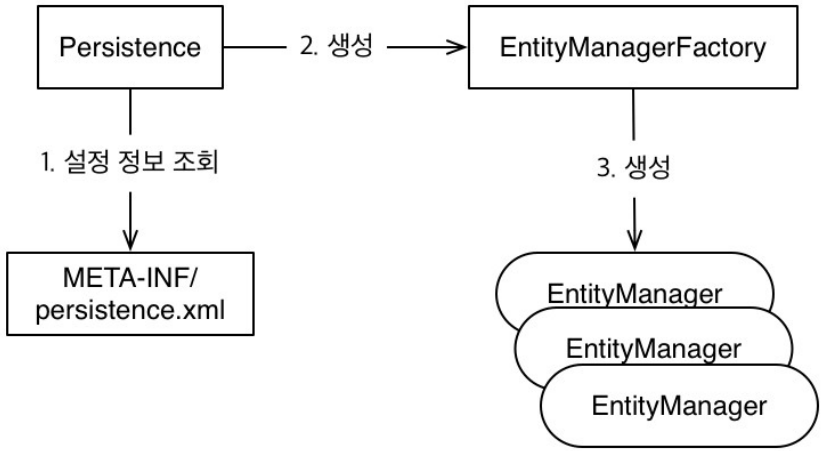
2.6.2 엔티티 매니저 설정

2) EntityManager 생성

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory( "jpabookmall" );
```

- JPA의 대부분 기능은 EntityManager가 제공하는 데, EntityManager를 생성해 주는 EntityManagerFactory 객체를 먼저 생성해 주어야 한다.

- Persistence 클래스가 EntityManagerFactory를 생성하는 데 이 때, persistence.xml 파일을 읽고 설정되어 있는 영속 유닛 이름으로 찾아 만들게 된다.



- EntityManagerFactory 뿐만 아니라 여러 기반 객체를 생성하고 커넥션 풀 등을 생성하고 설정하기 때문에 비용이 아주 큰 작업으로 **보통, 애플리케이션에서는 EntityManagerFactory는 하나만 생성하고 공유해서 사용한다.**

- **보통, 애플리케이션을 종료할 때** EntityManagerFactory도 종료한다.

```
emf.close();
```

2.6.2 엔티티 매니저 설정

2) EntityManager 생성

```
EntityManager em = emf.createEntityManager();
```

- EntityManager를 통해 데이터베이스의 CRUD 작업을 할 수 있다.
- EntityManager는 내부에 데이터소스를 유지하면서 데이터베이스와 통신한다.
- EntityManager를 가상의 데이터베이스로 생각할 수 도 있다.
- EntityManager는 데이터베이스 커넥션과 밀접한 관계가 있기 때문에 쓰레드 간의 공유나 재사용하면 않된다.
- 사용이 끝난 EntityManager는 반드시 종료 해야 한다.

```
em.close();
```

2.6 애플리케이션 작성하기

2.6.3 트랜잭션 관리

1) JPA에서는 항상 트랜잭션 안에서 CRUD 작업들을 해 주어야 한다. (예외 발생함)

2) 트랜잭션을 시작하려면 EntityManager에서 EntityTransaction 객체를 받아와야 한다.

```
EntityTransaction tx = em.getTransaction();
```

3) 트랜잭션 시작은 begin() 메서드를 호출한다.

```
tx.begin();
```

4) 그리고 비지니스 코드를 수행한다(보통 데이터베이스의 CRUD 작업들)

5) 정상적으로 비지니스 코드들이 수행되면 commit() 를 호출하여 실제 데이터베이스에 적용해야 한다.

```
tx.commit();
```

6) 실패 시(예외가 발생할 경우), 트랜잭션 rollback()을 하여 실패 전까지 실행되었던 CRUD작업을 취소할 수 있다.

2.6 애플리케이션 작성하기

2.6.4 비즈니스 로직 작성하기

1) Create(insert, 저장)

```
private static void insertLogic( EntityManager em ) {
    Book book = new Book();
    book.setNo( 1L );
    book.setTitle( "자바의 신" );
    book.setPrice( 20000 );

    // 저장
    em.persist( book );
}
```

- 엔티티(객체)를 저장하기 위해 EntityManager의 persist() 메서드를 사용한다.

- JPA는 전달된 엔티티(객체)의 어노테이션을 분석하여 sql문을 생성하고 데이터베이스에 전달한다.

2.6.4 비즈니스 로직 작성하기

2) Update(update, 수정)

```
private static void logicInsertAndUpdate( EntityManager em ) {
    Book book = new Book();
    book.setNo( 2L );
    book.setTitle( "곰브리치 세계사" );
    book.setPrice( 30000 );

    // 저장
    em.persist( book );

   // 가격 Update
   book.setPrice( 50000 );
}
```

- 단순히 엔티티 값의 변경으로 데이터베이스에 수정(Update)이 가능하다.

- EntityManager에 update() 같은 메서드는 없다.

2.6 애플리케이션 작성하기

2.6.4 비즈니스 로직 작성하기

3) Read(select, 1건 조회)

```
private static void logicFindOne( EntityManager em ) {
    // 조회
    Book book = em.find( Book.class, 1L );

    System.out.println( book );
}
```

- EntityManager의 find() 메서드를 사용한다.

- 찾을 엔티티 클래스 타입과 @Id로 데이터베이스 기본 키와 매핑한 식별자 값으로 엔티티 하나를 조회한다.

- 이 메서드를 호출하면 EntityManager는 select sql문을 생성하여 데이터베이스에 결과를 조회한다.

- 그리고 결과 값으로 엔티티를 생성해서 반환한다.

[실습]
1. 존재하지 않는 식별자 값으로 조회해 보기
2. 조회된 엔티티의 가격을 수정하고 데이터베이스에서 확인해 보기

2.6 애플리케이션 작성하기

2.6.4 비즈니스 로직 작성하기

3) Read(select, 목록조회)

```
private static void logicFindList( EntityManager em ) {
   // 목록조회(JPQL)
   TypedQuery<Book> query = em.createQuery( "select b from Book b", Book.class );
    List<Book> list = query.getResultList();

    for( Book book : list ) {
        System.out.println( book );
    }
}
```

- JPA에서는 엔티티 객체 중심으로 개발한다. (SQL를 사용하지 않음)
- 문제는 검색 쿼리이다. (검색 조건이 포함된 SQL를 사용해야 함)
- JPA에서는 JPQL이라는 쿼리 언어로 이 문제를 해결한다.

JPQL(Java Persitence Query Language)

- SQL를 추상화한 **객체 지향 쿼리 언어**
- SQL과 문법이 유사하다.
- JPQL은 엔티티 객체를 대상으로 한다. 즉 클래스와 필드를 대상으로 한다.(SQL은 데이터베이스의 테이블을 대상)
- JPQL은 데이터베이스의 테이블을 전혀 알지 못한다.
- JPQL을 사용하기 위해 create(JPQL, 반환 타입) 통해 쿼리 객체를 생성한 후, 조회 메서드를 호출한다.
- JPA가 JPQL을 적절한 SQL로 변환해서 데이터 베이스에서 데이터를 조회한다.

[실습]
검색된 엔티티 객체를 변경하고 결과를 알아본다.

2.6.4 비즈니스 로직 작성하기

4) Delete(delete, 삭제)

```
private static void logicFindOneAndDelete( EntityManager em ) {
    // 조회
    Book book = em.find( Book.class, 2L );

    // 삭제
    em.remove( book );
}
```

- EntityManager의 remove() 메서드를 사용한다.

- 이 메서드를 호출하면 EntityManager는 delete sql문을 생성하여 데이터베이스에서 삭제 한다.

JPA & Hibernate

3.1 엔티티 매니저 팩토리와 엔티티 팩토리

엔티티 매니저 팩토리

```
EntityManagerFactory  emf  =  Persistence.createEntityManagerFactory( "jpabookmall" );
```

- 엔티티 매니저를 만드는 공장

- 비용은 상당히 크다. 따라서 한 개만 만들어서 애플리케이션 전체에서 공유하도록 설계되어 있다.

- 반면에 공장에서 엔티티 매니저를 생성하는 비용은 거의 들지 않는다.

- 데이터베이스를 하나만 사용하는 애플리케이션은 일반적으로 EntityManagerFactroy 를 하나만 생성한다.

- 필요할 때마다 엔티티 매니저를 생성하면 된다.

엔티티 매니저

```
EntityManager  em  = emf.createEntityManager;
```

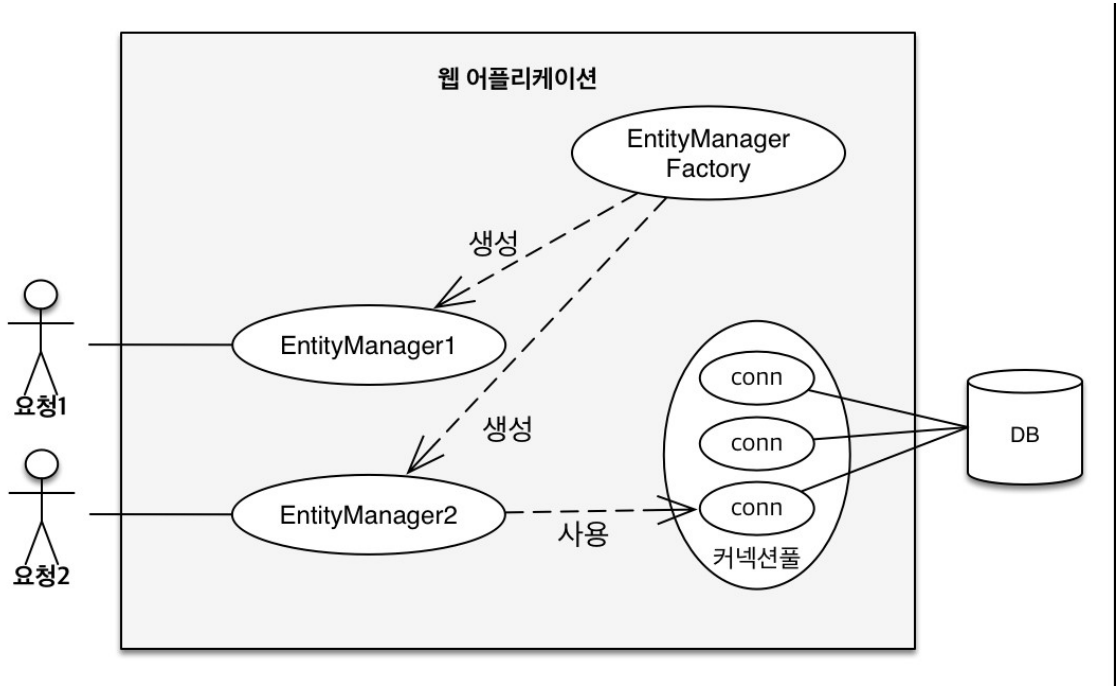
- 엔티티 객체를 관리하는 관리자

- 엔티티 매니저를 통해 테이블과 매핑된 엔티티를 실제 사용하게 된다. (저장, 수정, 삭제, 조회)

- 개발자 입장에서는 가상의 데이터베이스로 생각해도 무방하다.

3.1 엔티티 매니저 팩토리와 엔티티 팩토리

웹 애플리케이션 에서의 엔티티 매니저 팩토리와 엔티티 팩토리



- EntityManagerFactory 에서 다수의 EntityManager를 생성
- EntityManager는 트랜잭션을 시작할 때 커넥션을 얻는다.
- EntityManagerFactory는 여러 쓰레드가 동시에 접근해도 안전하다.
- EntityManager는 여러 쓰레드가 동시에 접근하면 동시성 문제가 발생하므로 쓰레드간에 절대 공유하면 안된다.

3.2 영속성 컨텍스트(Persistence Context)

3.2.1 엔티티를 영구 저장하는 환경(공간, 영역)

1) 엔티티 매니저를 통행 엔티티 객체를 저장하거나 조회하면 엔티티 매니저는 영속성 컨텍스트에 엔티티를 보관하며 관리하게 된다.

2) 도서 엔티티 객체를 데이터베이스에 저장한다고 표현되지만,

```
em.persist( book );
```

정확한 표현은 “엔티티 매니저로 도서 엔티티를 영속성 컨텍스트에 저장 한다”

3) 영속성 컨텍스트는 논리적 개념으로 엔티티 매니저가 생성될 때 하나 만들어 진다.

4) 영속성 컨텍스트는 엔티티 매니저를 통해 접근할 수 있고 관리할 수 있다.

3.2 영속성 컨텍스트(Persistence Context)

3.2.2 엔티티의 생명 주기

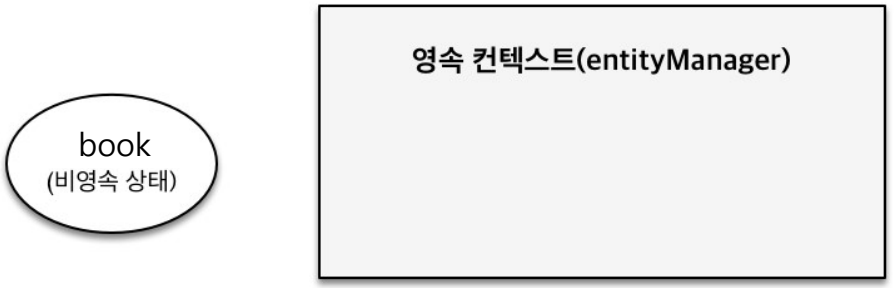
엔티티에는 4가지 상태가 존재한다.

1) 비영속 (new/transient)

```
Book book = new Book();
   book.setNo( 1L );
   book.setTitle( "자바의 신" );
   book.setPrice( 20000 );
```

- 엔티티 객체를 생성한 상태로 em.persist() 호출 전 상태

- 영속성 컨텍스트나 데이터베이스와 전혀 관계가 없는 상태



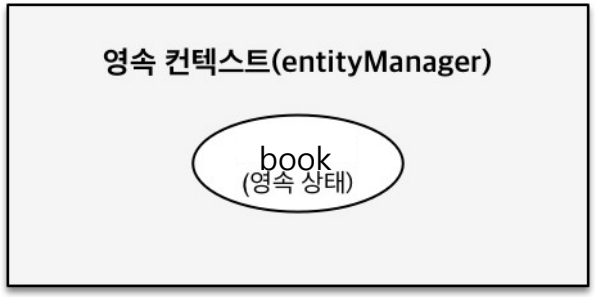
3.2 영속성 컨텍스트(Persistence Context)

2) 영속 (managed)

```
em.persist( book );
```

- 엔티티 매니저를 통해 영속성 컨텍스트에 저장된 상태

- 엔티티 가 영속성 컨텍스트에 의해 관리 되는 상태다.



- em.find() 또는 JPQL을 사용해서 조회된 엔티티도 영속성 상태가 된다.

```
Book book = em.find( Book.class, 1L );
```

3) 준영속 (detached)

```
em.detach( book );
```

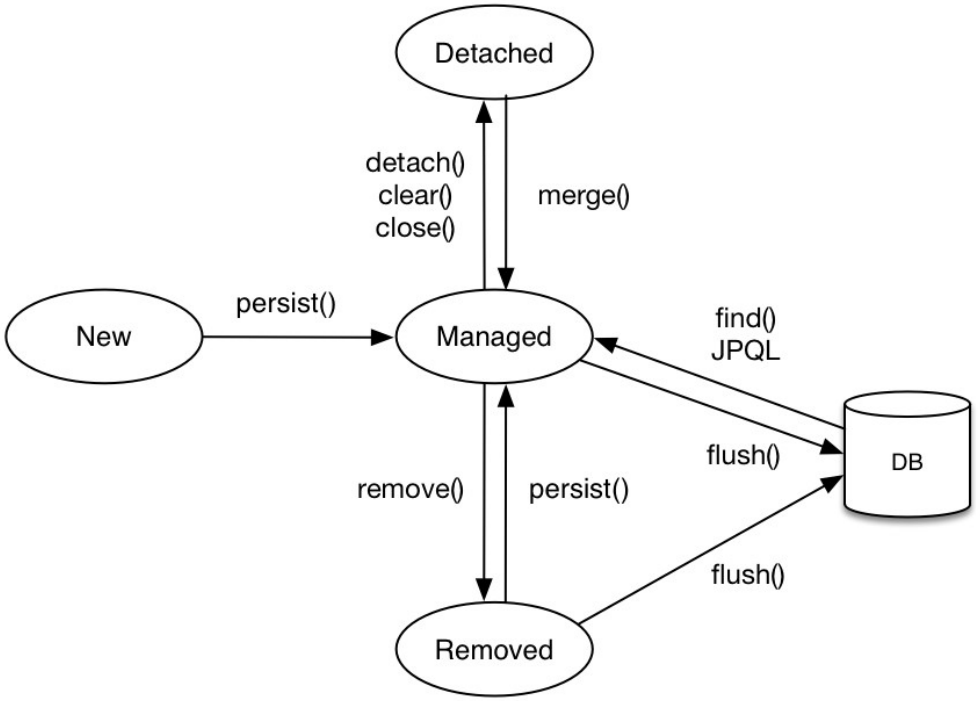
- 영속성 컨텍스트에 저장되었다가 분리된 상태
- 영속성 컨텍스트가 관리하던 영속성 상태의 엔티티를 더이상 관리하지 않으면 준영속 상태가 된다.
- detach() 호출하면서 영속성 상태의 엔티티를 넘겨주면 된다.
- em.close()를 호출해서 영속성 컨텍스트를 닫으면 컨텍스트 안의 모든 엔티티는 준영속 상태가 된다.
- em.clear()를 호출해 영속성 컨텍스트를 초기화 하면 컨텍스트 안의 모든 엔티티는 준영속 상태가 된다.

4) 삭제 (removed)

```
em.remove( book );
```

- 삭제된 상태로 엔티티를 영속성 컨텍스트와 데이터베이스에서 삭제한다.

엔티티의 생명주기



JPA & Hibernate

4.1 엔티티 매핑

4.1.1 @Entity

JPA를 사용해서 테이블과 매핑할 클래스는 @Entity 를 반드시 붙어야 한다. @Entity가 붙은 클래스는 JPA가 관리하는 것으로, 엔티티라 한다.

1) 속성

name

JPA에서 사용할 엔티티 이름을 지정한다. 보통 기본값인 클래스 이름을 사용한다.
만약 다른 패키지에 이름이 같은 엔티티 클래스가 있다면 이름을 지정해서 충돌하지 않도록 해야 한다.
설정하지 않으면 클래스 이름을 그대로 사용한다.

2) 적용 시 주의점

- 기본 생성자는 필수다. (파라미터가 없는 public 또는 protected 생성자)

- final 클래스, enum , interface , inner 클래스에는 사용할 수 없다.

- 저장할 필드에 final 을 사용하면 안 된다.

4.1 엔티티 매핑

4.1.2 @Table

엔티티와 매핑할 테이블을 지정한다. 생략하면 매핑한 엔티티 이름을 테이블 이름으로 사용한다.

1) 속성

name
매핑할 테이블 이름 (기본값은 엔티티의 이름)

schema
schema 기능이 있는 데이터베이스에서 schema 를 매핑

4.1.3 데이터베이스 스키마 자동 생성하기

JPA는 매핑 정보를 활용해서 데이터베이스의 스키마를 자동으로 생성하는 기능을 지원한다.
클래스의 매핑 정보를 보면 어떤 테이블에 어떤 컬럼을 사용하는지 알 수 있다. JPA는 이 매핑 정보와 데이터베이스 방언을 사용해서 데이터베이스 스키마를 생성한다.

1) hibernate.hbm2ddl.auto 속성을 추가하면 애플리케이션 실행 시점에 데이터베이스 테이블을 자동으로 생성한다.

2) 애플리케이션 실행 시점에 데이터베이스 테이블이 자동으로 생성되므로 개발자가 테이블을 직접 생성하는 수고를 덜 수 있다.

3) 하지만 스키마 자동 생성 기능이 만든 DDL은 운영 환경에서 사용할 만큼 완벽하지 않고 단, 개발 환경에서 사용하거나 매핑을 어떻게 해야 하는지 참고하는 정도로 사용한다.

4.1.4 키본 키(Primary Key) 매핑

1) @id

엔티티 클래스의 필드를 테이블의 기본 키(Primary key)에 매핑 한다. @Id가 사용된 필드를 **식별자 필드**라 한다.

2) 기본 키 생성 전략

- 직접 할당 : 기본 키를 애플리케이션에서 직접 할당한다.
- 자동 생성 : 대리 키 사용 방식

  IDENTITY : 기본 키 생성을 데이터베이스에 위임한다. (MySQL, PostgreSQL, SQL Server, DB2)에서 사용한다.

  SEQUENCE : 데이터베이스 시퀀스를 사용해서 기본 키를 할당한다. (Oracle, PostgreSQL, DB2, H2)

  TABLE : 키생성 테이블을 사용한다. (직접 운용한다.)

- MySQL의 경우 @GeneratedValue 로 적용할 수 있다.

  @GeneratedValue(strategy= GenerationType.IDENTIFY)

4.1 엔티티 매핑

4.1.5 필드와 컬럼 매핑

1) @Column

- 컬럼을 매핑한다

- 속성

name :
필드와 매핑할 테이블의 컬럼 이름 (기본값은 필드 이름)

nullable(DDL) :
null 값의 허용 여부를 설정한다. false 로 설정하면 DDL 생성시에 not null 제약조건이 붙는다. (기본값은 true)

length(DDL) :
문자 길이 제약조건, String 타입에만 사용한다. (기본값 255)

precision, scale(DDL) :
BigDecimal, BigInteger 에 사용. precision 은 소수점을 포함한 전체 자릿수를, scale 은 소수의 자릿수다.
참고로 double , float 타입에는 적용되지 않는다. 아주 큰 숫자나 정밀한 소수를 다루어야 할 때만 사용한다.

4.1.5 필드와 컬럼 매핑

2) @Enumerated

- 자바의 enum 타입을 매핑할 때 사용한다.

- 속성
  value :
  EnumType.ORDINAL : enum 순서를 데이터베이스에 저장
  EnumType.STRING : enum 이름을 데이터베이스에 저장

  기본값은 EnumType.ORDINAL 이다.

3) @Temporal

- 날짜 타입(java.util.Date , java.util.Calendar)을 매핑할 때 사용한다.

- 속성
  value :
  TemporalType.DATE : 날짜, 데이터베이스 date 타입과 매핑 (예: 2013-10-11)
  TemporalType.TIME : 시간, 데이터베이스 time 타입과 매핑 (예: 11:11:11)
  TemporalType.TIMESTAMP : 날짜와 시간, 데이터베이스 timestamp(datetime) 타입과 매핑
  (예: 2013-10-11 11:11:11)

4.1.5 필드와 컬럼 매핑

4) @Lob

- 데이터베이스 BLOB , CLOB 타입과 매핑한다.

- @Lob 에는 지정할 수 있는 속성이 없다. 대신에 매핑하는 필드 타입이 문자면 CLOB 으로 매핑하고 나머지는 BLOB 으로 매핑한다.
  CLOB : String , char[] , java.sql.CLOB
  BLOB : byte[] , java.sql.BLOB

3) @Transient

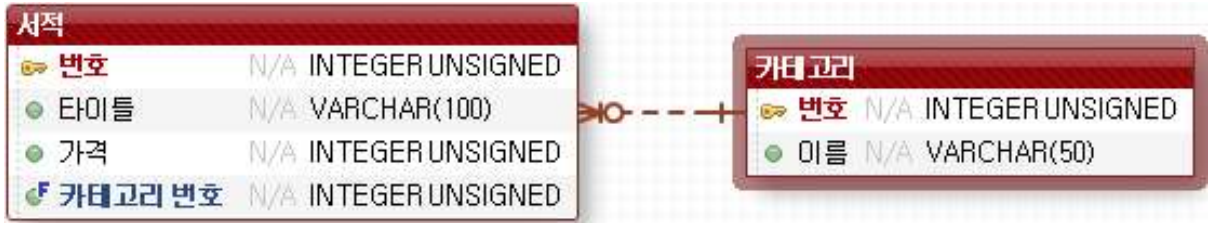
- 이 필드는 매핑하지 않는다. 따라서 데이터베이스에 저장하지 않고 조회하지도 않는다. 객체에 임시로 어떤 값을 보관하고 싶을 때 사용할 수 있다.

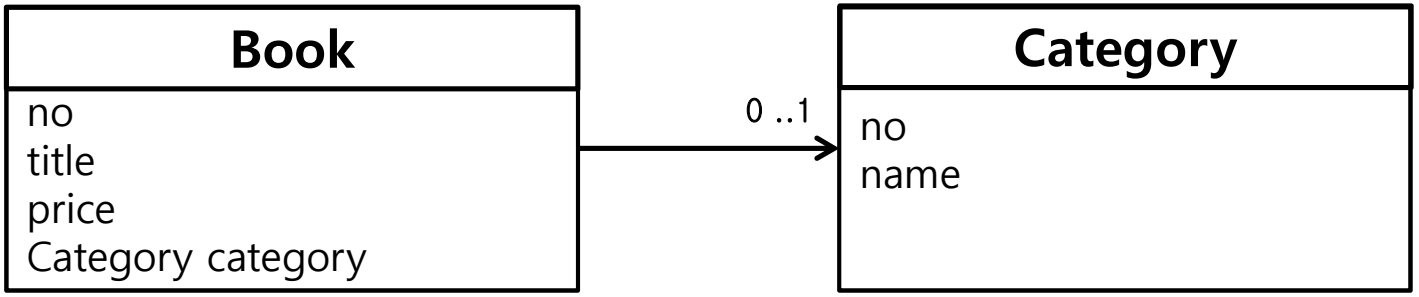
4.2 연관관계 매핑

4.2.1 연관관계 매핑이란?

- 비즈니스 엔티티들은 대부분 다른 엔티티와 연관관계가 있다.

예)





- 객체는 참조(주소)를 사용해서 관계를 맺고 테이블은 외래 키를 사용해서 관계를 맺는다.
- 이 둘은 완전히 다른 특징을 가진다.
- 객체 관계 매핑(ORM)에서 가장 어려운 부분이 바로 객체 연관관계와 테이블 연관관계를 매핑하는 일이다.

4.2 연관관계 매핑

4.2.1 연관관계 매핑이란?

1) 객체의 참조와 테이블의 외래 키를 매핑하는 것이 목표

2) 주요 용어 정리

- 방향(Direction):
서적 -> 카테고리 또는 카테고리 -> 서적 둘 중 한쪽만 참조하는 것을 **단방향 관계**라 하고,
서적 -> 카테고리, 카테고리 -> 서적 양쪽 모두 서로 참조하는 것을 **양방향 관계**라 한다.

방향은 객체관계에만 존재하고 테이블 관계는 항상 양방향이다.

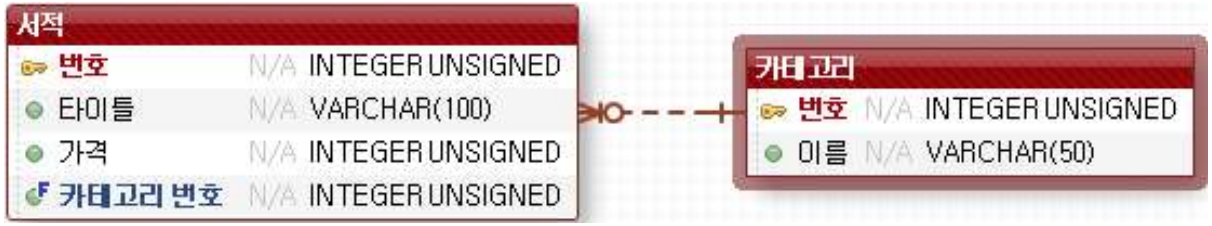
- 다중성(Multiplicity):
다대일(N:1), 일대다(1:N), 일대일(1:1), 다대다(N:N) 다중성이 있다.
예)
서적과 카테고리가 관계가 있을 때 여러 서적은 한 카테고리에 속하므로 서적과 카테고리는 다대일 관계다.
그럼 반대는?

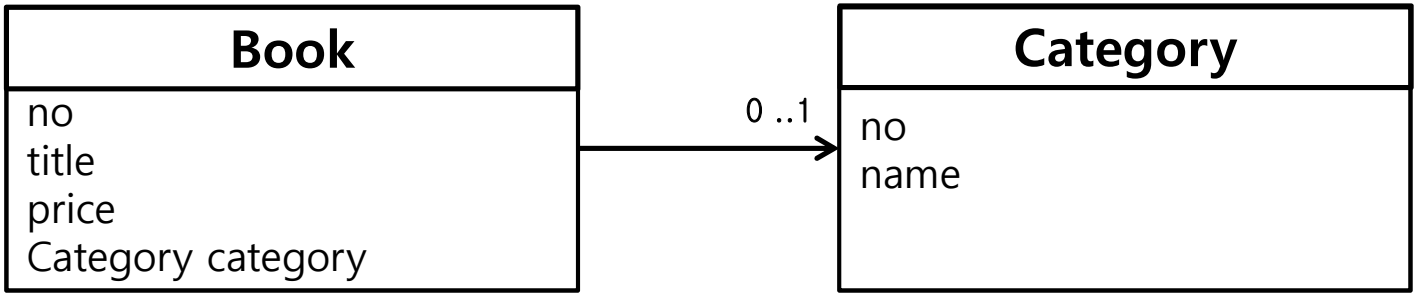
- 연관관계의 주인(owner):
객체를 양방향 연관관계로 만들면 연관관계의 주인을 정해야 한다.

4.2 연관관계 매핑

4.2.2 단방향 연관관계

연관관계 중에선 다대일(N:1) 단방향 관계를 가장 먼저 이해해야 한다.





- 서적과 카테고리가 있다.
- 서적은 하나의 카테고리에만 소속될 수 있다.
- 서적과 카테고리는 다대일 관계다.

4.2 연관관계 매핑

4.2.2 단방향 연관관계

1) 객체 연관관계

- 서적 객체는 Book.category 필드(멤버변수)로 카테고리 객체와 연관관계를 맺는다.

- 서적 객체와 카테고리 객체는 **단방향 관계**다.

- 서적은 Book.category 필드를 통해서 카테고리를 알 수 있다.

- 하지만, 반대로 카테고리는 서적을 알 수 없다.

예)
book -> category 의 조회는 book.getCatergory() 로 가능하지만,
반대 방향인 category -> book 를 접근하는 필드는 없다.

4.2.2 단방향 연관관계

2) 테이블 연관관계

- 서적 테이블은 category_no 외래 키로 카테고리 테이블과 연관관계를 맺는다.
- 서적 테이블과 카테고리 테이블은 **양방향 관계**다.
- 서적 테이블의 category_no 외래 키를 통해서 서적과 카테고리를 조인할 수 있고 반대로 카테고리와 서적도 조인할 수 있다.

예)

```
SELECT *
   FROM book b
     JOIN category c ON
b.category_no = c.no
```

```
SELECT *
   FROM category c
     JOIN book b ON  c.no =
b.category_no
```

4.2 연관관계 매핑

4.2.2 단방향 연관관계

3) 객체 연관관계와 테이블 연관관계의 가장 큰 차이

- 참조를 통한 연관관계는 언제나 단방향이다.

- 객체간에 연관관계를 양방향으로 만들고 싶으면 반대쪽에 도 필드를 추가해서 참조를 보관해야 한다.
  결국 연관관계를 하나 더 만들어야 한다. 이렇게 양쪽에서 서로 참조하는 것을 양방향 연관관계라 한다.
  (양방향 관계가 아니라 서로 다른 단뱡향 관계 2개다.)

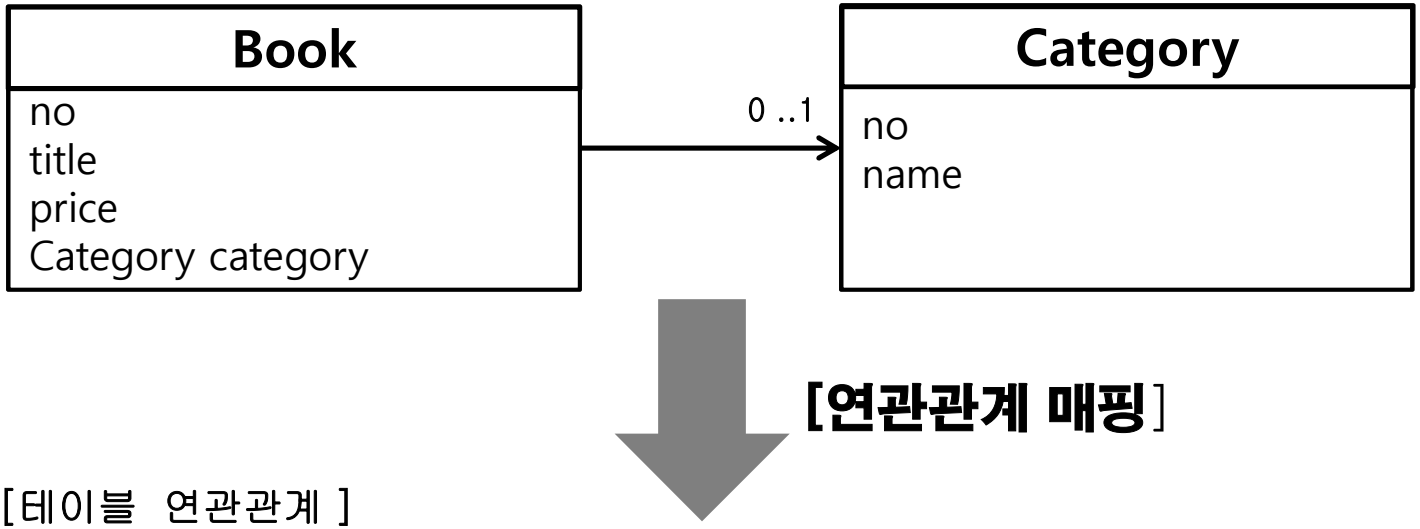
- 테이블은 외래 키 하나로 양방향으로 조인할 수 있다.

- 객체는 참조(주소)로 연관관계를 맺는다. 반면, 테이블은 외래 키로 연관관계를 맺는다.

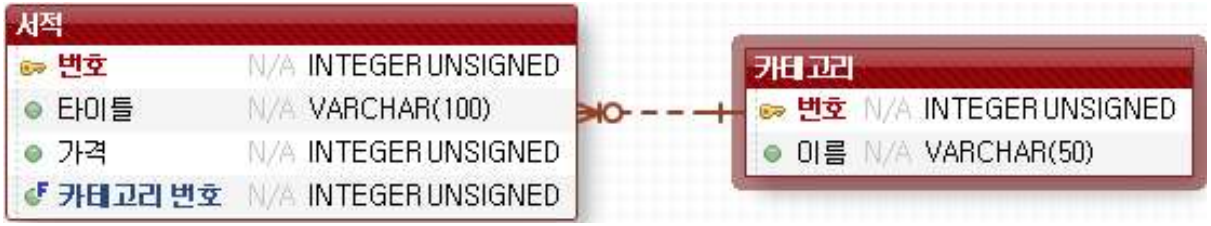
4.2.2 단방향 연관관계

4) 객체 관계 매핑하기 [실습]

[객체 연관관계]



[테이블 연관관계]



4.2 연관관계 매핑

4.2.2 단방향 연관관계

4) 객체 관계 매핑하기

[Book Entity]

```
@Entity
@Table( name = "book" )
public class Book {

   @Id
   @Column( name = "no" )
   @GeneratedValue ( strategy = GenerationType.IDENTITY )
   private Long no;

   @Column( name = "title" )
   private String title;

   @Column( name = "price" )
   private Integer price;

  private Category category;


}
```

[Category Entity]

```
@Entity
@Table( name="category" )
public class Category {

@Id
@Column( name="no" )
@GeneratedValue ( strategy = GenerationType.IDENTITY )
private Long no;

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

4.2 연관관계 매핑

4.2.2 단방향 연관관계

4) 객체 관계 매핑하기

```
@ManyToOne
@JoinColumn( name = "category_no" )
private Category category;
```

@ManyToOne :
이름 그대로 다대일(N:1) 관계라는 매핑정보다. 회원과 팀은 다대일 관계다.
연관관계를 매핑할 때 이렇게 다중성을 나타내는 어노테이션은 필수로 사용해야 한다.

@JoinColumn(name="category_no") :
조인 컬럼은 외래 키를 매핑할 때 사용한다.
name 속성에는 매핑할 외래 키 이름을 지정한다.

회원과 팀 테이블은 category_no 외래 키로 연관관계를 맺으므로 이 값을 지정하면 된다.
생략 시 다음 전략이 적용 된다.

필드명 + “_” + 참조하는 테이블의 기본 키(@Id) 컬럼명

4.2 연관관계 매핑

4.2.2 단방향 연관관계

5) 연관 관계 사용하기

- 저장 테스트

```
public static void testSave( EntityManager em ) {

    Category category = new Category();
    category.setName( "프로그래밍" );
    em.persist( category );

    Book book = new Book();
    book.setTitle( "자바의 신" );
    book.setPrice( 20000 );
    book.setCategory(category);

    // 저장
    em.persist( book );

}
```

- 쿼리를 확인해 본다.

- 테이블을 직접 확인해서 외래 키 상태를 확인 해본다.

- 주의할 점은 “엔티티 저장 시, 연관된 모든 엔티티도 영속 상태” 여야 한다.

4.2.2 단방향 연관관계

5) 연관 관계 사용하기

- 조회 테스트1 (객체 그래프 탐색)

```
public static void findBook( EntityManager em ) {

   Book book = em.find( Book.class, 1L );
   System.out.println( book );

   Category category = book.getCategory();
   System.out.println( category );

}
```

- book.getCategory() 을 사용해서 book와 연관된 category 엔티티를 조회할 수 있다.

- 이처럼 객체를 통해 연관된 엔티티를 조회하는 것을 **객체 그래프 탐색** 이라 한다.

4.2.2 단방향 연관관계

5) 연관 관계 사용하기

- 조회 테스트2 (JPQL Join)

```
public static void findBook2( EntityManager em ) {

   String jpql =  "select b from Book b join b.category c where b.title = ?1";
   TypedQuery<Book> query = em.createQuery( jpql, Book.class );

    query.setParameter( 1, "Spring in Action" );
    List<Book> list = query.getResultList();

    for( Book book : list ) {
       System.out.println( book );
    }
}
```

실행된 SQL과 JPQL을 비교하면 JPQL은 객체(엔티티)를 대상으로 하고 SQL보다 간결하다.

4.2 연관관계 매핑

4.2.2 단방향 연관관계

5) 연관 관계 사용하기

- 수정

```
public static void updateBooks( EntityManager em ) {
   // no = 4L
    Category category = new Category();
    category.setName( "IT" );
    em.persist( category );

    Book book1 = em.find( Book.class, 1L );
    book1.setCategory(category);

    Book book2 = em.find( Book.class, 2L );
    book2.setCategory(category);

    Book book3 = em.find( Book.class, 3L );
    book3.setCategory(category);
}
```

- em.update() 같은 메서드가 없다.

- 단순히 불러온 엔티티의 값만 변경해두면 트랜잭션을 커밋할 때 플러시가 일어나면서 **변경 감지** 기능이 작동한다. 그리고 변경사항을 데이터베이스에 자동으로 반영한다.

- 연관관계 수정에서도 참조하는 대상만 변경하면 나머지는 JPA가 자동으로 처리한다.

4.2.2 단방향 연관관계

5) 연관 관계 사용하기

- 연관관계 제거

```
public static void removeRelation( EntityManager em ) {

   Book book = em.find( Book.class, 1L );
   book.setCategory( null );

}
```

연관관계를 널(null)로 설정했다.

4.2 연관관계 매핑

4.2.2 단방향 연관관계

5) 연관 관계 사용하기

– 연관된 엔티티 제거

```
public static void removeCategory( EntityManager em ) {
    Category category = em.find( Category.class, 4L );
    em.remove( category );
}
```

연관된 엔티티를 삭제하려면 기존에 있던 연관관계를 먼저 제거하고 삭제해야 한다. 그렇지 않으면 외래 키 제약조건으로 인해, 데이터베이스에서 오류가 발생한다.

```
public static void removeCategory( EntityManager em ) {

    Book book1 = em.find( Book.class, 1L );
    em.remove( book1 );

    Book book2 = em.find( Book.class, 2L );
    em.remove( book2 );

    Book book3 = em.find( Book.class, 3L );
    em.remove( book3 );

    Category category = em.find( Category.class, 4L );
    em.remove( category );

}
```

4.2 연관관계 매핑

[실습 과제]

MySite 4 JPA + Hibernate 적용 하기

User, Guestbook, Board 엔티티 매핑하기
User 와 Board 엔티티 연관 관계 매핑 하기

[참고 JPQL]

1. 많은 데이타 업데이트 및 삭제 (벌크 연산)

예)

```
String psql = "delete from Product p where p.price < :price";

 int resultCount = em.createQuery( psql ).setParameter( “price", 1000 ).executeUpdate();
```

2. Like 검색 - SQL문과 같다

```
select m from Member m where m.username like '%원%'
```

4.2 연관관계 매핑

[참고 JPQL]

3. 페이징 처리

- 페이징 처리용 SQL을 작성하는 일은 지루하고 반복적이다.
- 더 큰 문제는 데이터베이스마다 페이징을 처리하는 SQL 문법이 다르다는 점이다.

JPA는 페이징을 다음 두 API로 추상화했다.

setFirstResult(int startPosition) : 조회 시작 위치 (0부터 시작한다.)
setMaxResults(int maxResult) : 조회할 데이터 수

```
TypedQuery<Member> query =
        em.createQuery( "SELECT m FROM Board b ORDER BY b.regDate DESC", Board.class);

query.setFirstResult( ( page-1) * 10 );
query.setMaxResults( 10 );

List<Board> list = query.getResultList();
```

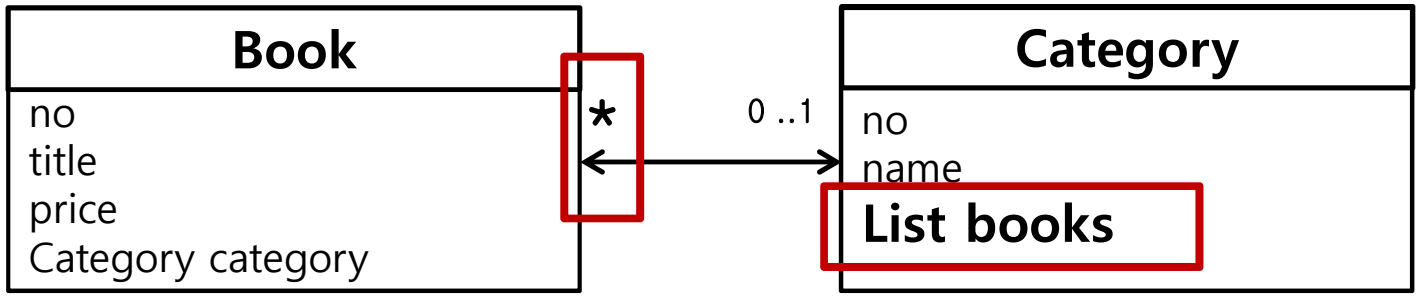
4.2 연관관계 매핑

4.2.3 양방향 연관관계

1) 객체 연관 관계

반대 방향인 카테고리에서 서적으로 관계를 추가하면 양방향 관계가 완성 된다.

[객체 연관관계]



- Book과 Category은 다대일(N:1) 관계다.

- 반대로, **카테고리에서 서적은 일대다(1:N) 관계**다.

- 일대다 관계는 여러 건과 연관관계를 맺을 수 있으므로 **컬렉션**을 사용해야 한다.
  Category.books 를 List 컬렉션으로 추가했다.

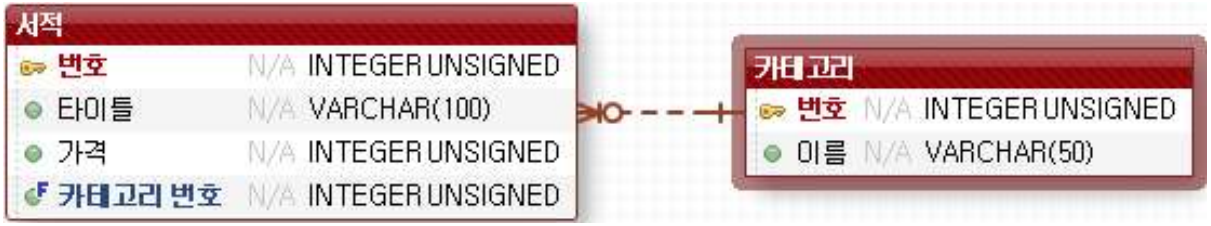
  객체 연관관계 정리
  서적 -> 카테고리 (Book.category), 카테고리 -> 서적 (Category.books)

4.2 연관관계 매핑

4.2.3 양방향 연관관계

2) 테이블 연관 관계

[테이블 연관관계]



- 두 테이블의 연관관계는 **외래 키 하나만으로 양방향 조회가 가능**하므로 처음부터 양방향 관계다.

- 데이터베이스에 추가할 내용이 전혀 없다.

- Book JOIN Category 도 가능하고 Category JOIN Book 도 가능하다.

4.2 연관관계 매핑

4.2.3 양방향 연관관계

3) 객체 관계 매핑하기

[Book Entity]

```
@Entity
@Table( name = "book" )
public class Book {

   @Id
   @Column( name = "no" )
   @GeneratedValue ( strategy = GenerationType.IDENTITY )
   private Long no;

   @Column( name = "title" )
   private String title;

   @Column( name = "price" )
   private Integer price;

  private Category category;


}
```

[Category Entity]

```
@Entity
@Table( name="category" )
public class Category {

@Id
@Column( name="no" )
@GeneratedValue ( strategy = GenerationType.IDENTITY )
private Long no;

@Column( name="name" )
private String name;

private List<Book> books;
```

4.2.3 양방향 연관관계

3) 객체 관계 매핑하기

```
@OneToMany( mappedBy = "category“ )
private List<Book> books = new ArrayList<Book>();
```

- 카테고리와 서적은 일대다 관계다. 따라서 컬렉션인 List<Book> books 를 추가

- 일대다 관계를 매핑하기 위해 **@OneToMany 매핑 정보를 사용**

- **mappedBy 속성은 양방향 매핑일 때 사용**하는데 **반대쪽 매핑의 필드 이름을 값**으로 주면 된다.
  반대쪽 매핑이 Book.category 이므로 category 를 값으로 주었다.

4.2.3 양방향 연관관계

4) 일대다 컬렉션 조회하기 (일대다 방향으로 객체 그래프 탐색)

```
public static void testFindBooksByCategory( EntityManager  em ) {

   Category category = em.find( Category.class, 1L );
   System.out.println( category );

   List<Book> list = category.getBooks();
   for(Book book : list ) {
      System.out.println(  book  );
   }

}
```

4.2 연관관계 매핑

4.2.3 양방향 연관관계

5) 연관관계의 주인(Owner)

- @OneToMany 만 있으면 되지 **mappedBy** 는 왜 필요할까?

- 서로 다른 단방향 연관관계 2개를 애플리케이션 로직으로 잘 묶어서 양방향인 것처럼 보이게 할 뿐, 양방향 연관 관계는 존재하지 않는다.

- 테이블은 외래 키 하나만으로 양방향 연관 관계를 맺고 **외래 키 하나로 두 테이블의 연관관계를 관리**한다.

객체 연관관계
서적 -> 카테고리 연관관계 1개(단방향)
카테고리 -> 서적 연관관계 1개(단방향)

테이블 연관관계
서적 <-> 카테고리의 연관관계 1개(양방향)

- **엔티티를 양방향 연관관계로 설정하면 객체의 참조는 둘인데 외래 키는 하나인 차이가 발생**하는데 둘 중 어떤 관계를 사용해서 외래 키를 관리해야 할까?

- JPA에서는 **두 객체 연관관계 중 하나를 정해서 테이블의 외래 키를 관리해야 하는데 이것을 연관관계의 주인**이라 한다.

4.2 연관관계 매핑

4.2.3 양방향 연관관계

5) 양방향 매핑의 규칙 – 연관관계의 주인(Owner)

- 두 연관관계 중 하나를 연관관계의 주인으로 정해야 한다(**양방향 연관관계 매핑 시 규칙**)

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

- 어떤 연관관계를 주인으로 정할지는 mappedBy 속성을 사용하면 된다.
  1. 주인은 mappedBy 속성을 사용하지 않는다.
  2. 주인이 아니면 mappedBy 속성을 사용해서 속성의 값으로 연관관계의 주인을 지정해야 한다.

[예제]
Member.team , Team.members 둘 중 어떤 것을 연관관계의 주인으로 정해야 하는 가?

- 연관관계의 주인을 정한다는 것은 사실 외래 키 관리자를 선택하는 것이다.

- 서적(book) 테이블에 있는 category_no 외래 키를 관리할 관리자를 선택해야 한다.

  1. Book 엔티티에 있는 Book.category 를 주인으로 선택하면 자기 테이블에 있는 외래 키를 관리하면 된다. ‘
  2. Category 엔티티에 있는 Category.books 를 주인으로 선택하면 물리적으로 전혀 다른 테이블의 외래 키를 관리해야 한다.

- **연관 관계의 주인은 외래 키가 있는 곳으로 정해야 한다.**

4.2.3 양방향 연관관계

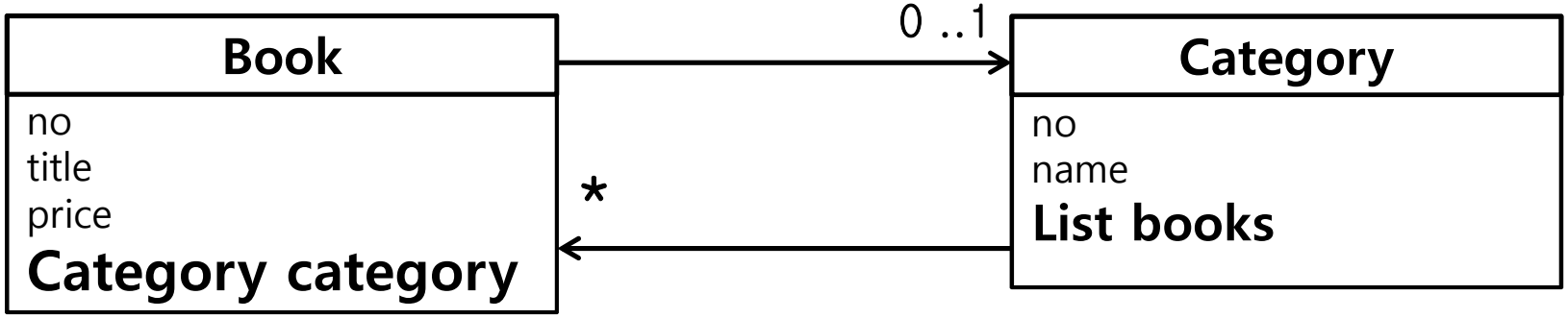
5) 양방향 매핑의 규칙 – 연관관계의 주인(Owner)

- 연관 관계의 주인은 외래 키가 있는 곳으로 정해야 한다.
- 따라서, 외래 키는 book 테이블이 가지고 있으므로 주인은 Book.category 가 된다.

주인은 mappedBy 속성을 사용할 수 없으므로 주인이 아닌 Category.books에 사용하여 주인이 아님을 나타내고 속성값으로 연관관계의 주인인 category를 표시해 주면 된다.

```
@OneToMany( mappedBy = "category“ )
private List<Book> books = new ArrayList<Book>();
```

진짜 매핑 – 연관관계의 주인(Book.category)



가짜 매핑 – 주인의 반대편(Category.books)

4.2.3 양방향 연관관계

5) 양방향 매핑의 규칙 - 연관관계의 주인(Owner) 정리

- 연관관계의 주인만 데이터베이스 연관관계와 매핑되고 외래 키를 관리할 수 있다.

- 주인이 아닌 반대편(inverse, non-owning side)은 읽기만 가능하고 외래 키를 변경하지는 못한다.

참고:
데이터베이스 테이블의 다대일, 일대다 관계에서는 항상 다 쪽이 외래 키를 가진다. 따라서 다 쪽인 @ManyToOne 는 항상 연관관계의 주인이 되므로 mappedBy 를 설정할 수 없다. 따라서 @ManyToOne에는 mappedBy 속성이 없다.

4.2 연관관계 매핑

4.2.3 양방향 연관관계

7) 양방향 연관 관계 저장

```
public static void testSave( EntityManager em ) {

    // 카테고리1 저장
    Category category1 = new Category();
    category1.setName( "카테고리1" );
    em.persist( category1 );

    // 책1
    Book book1 = new Book();
    book1.setTitle( "책1" );
    book1.setPrice( 1000 );
    book1.setCategory( category1 );
    em.persist( book1 );

    // 책2
    Book book2 = new Book();
    book2.setTitle( "책2" );
    book2.setPrice( 1000 );
    book2.setCategory( category1 );
    em.persist( book2 );

}
```

- 데이터베이스에서 book 테이블을 조회 해보자.

- 양방향 연관관계는 연관관계의 주인이 외래 키를 관리한다. 따라서 주인이 아닌 방향은 값을 설정하지 않아도 데이터베이스에 외래 키 값이 정상 입력된다.

4.2.3 양방향 연관관계

7) 양방향 연관 관계의 주의점

```
public static void testSaveNonOwner( EntityManager em ) {
    // 책1
    Book book1 = new Book();
    book1.setTitle( "책1" );
    book1.setPrice( 1000 );
    em.persist( book1 );

    // 책2
    Book book2 = new Book();
    book2.setTitle( "책2" );
    book2.setPrice( 1000 );
    em.persist( book2 );


    // 카테고리1 저장
    Category category1 = new Category();
    category1.setName( "카테고리1" );
    category1.getBooks().add( book1 );
    category1.getBooks().add( book2 );
    em.persist( category1 );
}
```

- 양방향 연관관계를 설정하고 가장 흔히 하는 실수다
- 연관관계의 주인에는 값을 입력하지 않고, 주인이아닌 곳에만 값을 입력하기 때문에 발생한다.
- 데이터베이스에 외래 키 값이 정상적으로 저장되지 않으면 이것부터 의심해보자.

4.2 연관관계 매핑

4.2.3 양방향 연관관계

8) 순수 객체의 관계까지 고려한 양방향 연관관계

- 객체 관점에서 양쪽 방향에 모두 값을 입력해 주는 것이 가장 안전하다.
- 양쪽 방향 모두 값을 입력하지 않으면 JPA를 사용하지 않는 순수한 객체 상태에서 심각한 문제가 발생할 수 있다.

```
public static void testSave( EntityManager em ) {

    // 카테고리1 저장
    Category category1 = new Category();
    category1.setName( "카테고리1" );
    em.persist( category1 );

    // 책1
    Book book1 = new Book();
    book1.setTitle( "책1" );
    book1.setPrice( 1000 );
    book1.setCategory( category1 );            //연관관계 설정 book1 -> category1
    book1.getCategory().add( book1 );          //연관관계 설정 category1 -> book1
    em.persist( book1 );

    // 책2
    Book book2 = new Book();
    book2.setTitle( "책2" );
    book2.setPrice( 1000 );
    book2.setCategory( category1 );           //연관관계 설정 book2 -> category1
    book2.getCategory().add( book2 );         //연관관계 설정 category1 -> book2
    em.persist( book2 );

}
```

4.2.3 양방향 연관관계

9) 순수 객체의 관계까지 고려한 양방향 연관관계 – 리팩토링

```
public void setCategory(Category category) {

    this.category = category;
    if( category == null ) {
        return;
    }

    category.getBooks().add( this );
}
```

Book 클래스를 다음과 같이 수정한 후,

```
book1.getCategory().add( book1 );
book2.getCategory().add( book2 );
```

코드를 없앤 후, 테스트해 보자.

4.2 연관관계 매핑

4.2.3 양방향 연관관계

10) 순수 객체의 관계까지 고려한 양방향 연관관계 – 리팩토링

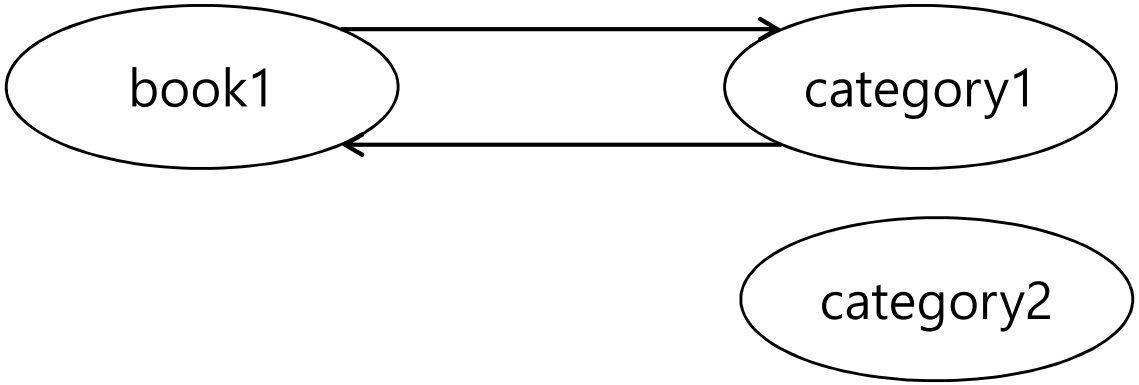
다음 코드를 실행 시킨 후, 앞의 리팩토링 코드의 버그를 찾아보고 안전하게 수정하여 보자.

```
public static void testSaveBug( EntityManager em ) {

    // 카테고리1 저장
    Category category1 = new Category();
    category1.setName( "카테고리1" );
    em.persist( category1 );

    // 카테고리2 저장
    Category category2 = new Category();
    category2.setName( "카테고리2" );
    em.persist( category2 );

    // 책1
    Book book1 = new Book();
    book1.setTitle( "책1" );
    book1.setPrice( 1000 );
    em.persist( book1 );

    book1.setCategory( category1 );
    book1.setCategory( category2 );

}
```

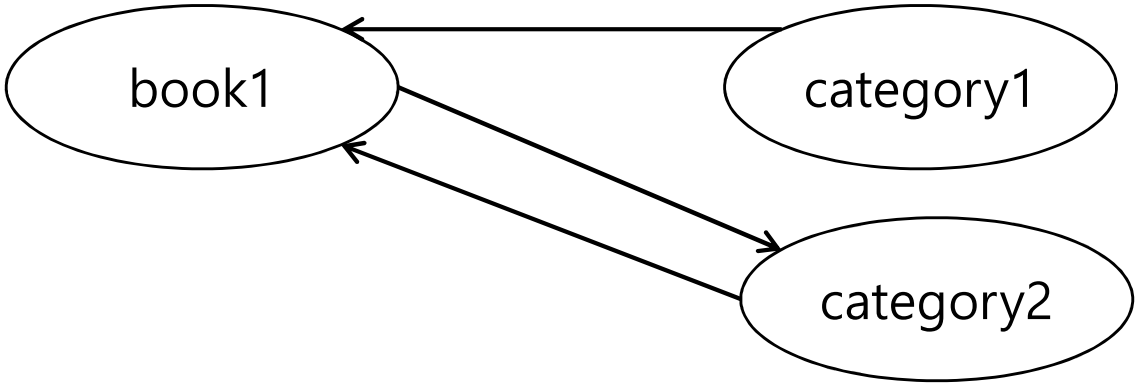
4.2.3 양방향 연관관계

10) 순수 객체의 관계까지 고려한 양방향 연관관계 – 리팩토링

[버그]
book1.setCategory(category1); 실행 시



book1.setCategory(category2); 실행 시



4.2 연관관계 매핑

4.2.3 양방향 연관관계

10) 순수 객체의 관계까지 고려한 양방향 연관관계 – 리팩토링

[실습 과제]
앞의 버그를 수정하기 위해 새 카테고리와 연관을 맺기 전에 기존 카테고리에서 자신을 제거해야 한다.
직접 수정해 보자.

4.2 연관관계 매핑

4.2.4 양방향 연관관계 정리

1) 단방향 매핑과 비교해서 양방향 매핑은 복잡하다.
   - 연관관계의 주인도 정해야 한다.
   - 두 개의 단방향 연관 관계를 양방향으로 만들기 위해 로직도 잘 관리해야 한다.

2) 연관관계가 하나인 단방향 매핑은 언제나 연관관계의 주인이다.

3) 양방향은 여기에 주인이 아닌 연관관계를 하나 추가했을 뿐이다.

4) 단방향과 비교해서 **양방향의 장점은 반대방향으로 객체 그래프 탐색 기능이 추가된 것** 뿐이다.

```
book.getCategory();  // 서적 -> 카테고리
category.geBooks(); // 카테고리 -> 서적 [ 양방향 매핑으로 추가된 기능 ]
```

5) 주인의 반대편은 mappedBy 로 주인을 지정해야 한다. 그리고 주인의 반대편은 단순히 보여주는 일(객체 그래프 탐색)만 할 수 있다.

[정리]

1. 단방향 매핑만으로 테이블과 객체의 연관관계 매핑은 이미 완료되었다.

2. 단방향을 양방향으로 만들면 반대방향으로 객체 그래프 탐색 기능이 추가된다.

3. 양방향 연관관계를 매핑 하려면 객체에서 양쪽 방향을 모두 관리해야 한다.

JPA & Hibernate

5.1 객체 지향 쿼리 소개

5.1.1 EntityManager.find() 메서드

1) 식별자로 엔티티 하나를 조회할 수 있다.

2) 조회한 엔티티에 객체 그래프 탐색을 사용하면 연관된 엔티티들을 찾을 수 있다.

식별자로 조회 : EntityManager.find()
객체 그래프 탐색 : (예) a.getB().getC()

5.1.2 현실적이고 복잡한 검색 방법이 필요

1) 식별자 조회 + 객체 그래프 탐색 만으로는 실제 애플리케이션 개발이 어려움

2) 데이터는 데이터베이스에 있으므로 SQL로 필요한 내용을 최대한 걸러서 조회해야 한다.

3) ORM을 사용하면 데이터베이스 테이블이 아닌 엔티티 객체를 대상으로 개발하므로 검색도 테이블이 아닌 엔티티 객체를 대상으로 하는 방법이 필요

4) **JPQL은 이런 문제를 해결하기 위해 만들어졌다.**

5.1 객체 지향 쿼리 소개

5.1.3 JPQL 의 특징

1) 테이블이 아닌 객체를 대상으로 검색하는 객체 지향 쿼리다.

2) SQL을 추상화해서 특정 데이터베이스 SQL에 의존하지 않는다.

3) SQL이 데이터베이스 테이블을 대상으로 하는 데이터 중심의 쿼리라면 JPQL은 엔티티 객체를 대상으로 하는 객체 지향 쿼리다.

4) JPQL을 사용하면 JPA는 이 JPQL을 분석한 다음 적절한 SQL을 만들어 데이터베이스를 조회한다.

5) 조회한 결과로 엔티티 객체를 생성해서 반환한다.

6) 처음 보면 SQL로 오해할 정도로 문법이 비슷하기 때문에 SQL에 익숙한 개발자는 몇 가지 차이점만 이해하면 쉽게 적응할 수 있다.

5.1 객체 지향 쿼리 소개

5.1.4 JPA가 지원하는 다양한 검색 방법

1) JPA 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.

2) JPA 공식 지원 기술은 아니지만 많이 사용하고 있는 기술

- QueryDSL:
  Criteria 쿼리처럼 JPQL을 편하게 작성하도록 도와주는 빌더 클래스 모음, 비표준 오픈소스 프레임워크다.

- JDBC 직접 사용, MyBatis 같은 SQL 매퍼 프레임워크 사용:
  필요하면 JDBC를 직접 사용할수 있다.

Criteria나 QueryDSL은 JPQL을 편하게 작성하도록 도와주는 빌더 클래스일 뿐이다. 따라서 JPQL을 이해해야 나머지도 이해할 수 있다.

5.2 JPQL

5.2.1 기본 문법과 쿼리 API

JPQL도 SQL과 비슷하게 **SELECT, UPDATE, DELETE 문을 사용**할 수 있다. 참고로 엔티티를 저장할 때는 em.persist() 메서드를 사용하면 되므로 **INSERT 문은 없다.**

1) SELECT 문

```
SELECT u FROM User AS u WHERE u.email = 'kickscar@gmail.com'
```

대소문자 구분
엔티티와 속성은 대소문자를 구분한다. 예를 들어 User, email 은 대소문자를 구분한다. 반면에 SELECT , FROM , AS 같은 JPQL 키워드는 대소문자를 구분하지 않는다.

엔티티 이름
JPQL에서 사용한 Member 는 클래스 명이 아니라 엔티티 명이다. 엔티티 명은 @Entity(name="XXX") 로 지정할 수 있다.
엔티티 명을 지정하지 않으면 클래스 명을 기본값으로 사용한다.
기본값인 클래스 명을 엔티티 명으로 사용하는 것을 추천한다.

별칭은 필수
User AS u 를 보면 UIser에 u라는 별칭을 주었다. JPQL은 별칭을 필수로 사용해야 한다. 따라서 다음 코드처럼 별칭 없이 작성하면 잘못된 문법이라는 오류가 발생한다.

```
SELECT email FROM User AS u
```

AS 는 생략할 수 있다.

5.2 JPQL

5.2.1 기본 문법과 쿼리 API

2) TypedQuery 와 Query

- 작성한 JPQL을 실행하려면 쿼리 객체를 만들어야 한다.

- 쿼리 객체는 TypeQuery 와 Query 가 있는데 반환할 타입을 명확하게 지정할 수 있으면 TypeQuery 객체를 사용하고, 반환 타입을 명확하게 지정할 수 없으면 Query 객체를 사용하면 된다.

- TypedQuery 를 사용

```
public static void testTypedQuery( EntityManager em ) {
    TypedQuery<Category> query = em.createQuery( "select c from Category c", Category.class );
    List<Category> list = query.getResultList();

    for( Category category : list ) {
        System.out.println( category );
    }
}
```

- Query 사용

```
public static void testQuery( EntityManager em ) {
   Query query = em.createQuery( "select c.no, c.name from Category c" );
   List list = query.getResultList();
   for( Object object : list ) {
      Object[] results = (Object[]) object;
      for( Object result : results ) {
         System.out.println( result );
      }
   }
}
```

5.2 JPQL

5.2.1 기본 문법과 쿼리 API

3) 결과 조회

다음 메서드들을 호출하면 실제 쿼리를 실행해서 데이터베이스를 조회한다.

query.getResultList() :

- 결과를 리스트로 반환한다.
- 만약 결과가 없으면 빈 컬렉션을 반환한다.

query.getSingleResult() :

- 결과가 정확히 하나일 때 사용한다.
- 결과가 없으면 javax.persistence.NoResultException 예외가 발생한다.
- 결과가 1개보다 많으면 javax.persistence.NonUniqueResultException 예외가 발생 한다.
- getSingleResult() 는 결과가 정확히 1개가 아니면 예외가 발생한다는 점에 주의해야 한다.

5.2.1 기본 문법과 쿼리 API

4) 파라미터 바인딩 – 이름 기준 파라미터

이름 기준 파라미터는 파라미터를 이름으로 구분하는 방법이다. 이름 기준 파라미터는 앞에 : 를 사용한다.

```
public static void testNamedParameter( EntityManager em ) {
   TypedQuery<Category> query = em.createQuery( "select c from Category c where name = :name", Category.class );
   query.setParameter( "name", "Java Programming" );

   List<Category> list = query.getResultList();
   for( Category category : list ) {
      System.out.println( category );
   }
}
```

5.2.1 기본 문법과 쿼리 API

4) 파라미터 바인딩 – 위치 기준 파라미터

위치 기준 파라미터를 사용하려면 ? 다음에 위치 값을 주면 된다. 위치 값은 1부터 시작한다.

```
public static void testNamedParameter( EntityManager em ) {
   TypedQuery<Category> query = em.createQuery( "select c from Category c where name = ?1", Category.class );
   query.setParameter( 1, "Java Programming" );

   List<Category> list = query.getResultList();
   for( Category category : list ) {
      System.out.println( category );
   }
}
```

위치 기준 파라미터 방식보다는 **이름 기준 파라미터 바인딩 방식을 사용하는 것이 더 명확하다.**

[주의]

```
//파라미터 바인딩 방식을 사용하지 않고 직접 JPQL을 만들면 위험하다. (SQL Injection)
"select  u  from  User  u where u.email = '" + email + "'"
```

5.2.1 기본 문법과 쿼리 API

5) 프로젝션(Projection)

- SELECT 절에 조회할 대상을 지정하는 것을 프로젝션이라 하고
- [SELECT {프로젝션 대상} FROM] 으로 대상을 선택한다.
- 프로젝션 대상은 Entity, Embeded 타입, Scalar 타입이 있다. 스칼라 타입은 숫자, 문자 등 기본 데이터 타입을 뜻한다.

[1] 엔티티 프로젝션

```
select u from User u              // 회원
select b from Book b              // 서적
select c from Category c          // 카테고리
```

- 쉽게 생각하면 원하는 객체를 바로 조회한 것
- 이렇게 **조회한 엔티티는 영속성 컨텍스트에서 관리된다.**

5.2.1 기본 문법과 쿼리 API

[2] 스칼라 타입 프로젝션

- 숫자, 문자, 날짜와 같은 기본 데이터 타입들을 스칼라 타입이라 한다.

- 전체 회원의 이름을 조회

```
List<String> userNames = em.create( “select u.name from User u”, String.class ).getResultList();
```

- 중복을 제거하기 위해서 distinct 도 사용할 수 있다.

5.2 JPQL

5.2.1 기본 문법과 쿼리 API

[3] 여러 값 조회

- 엔티티를 대상으로 조회하면 편리하겠지만, 꼭 필요한 데이터들만 선택해서 조회해야 할 때도 있다.
- 프로젝션에 여러 값을 선택하면 TypeQuery 를 사용할 수 없고 대신에 Query 를 사용해야 한다.
- 앞의 testQuery는 다음과 같이 줄여 사용하면 조금 편하다.

```
public static void testQuery2( EntityManager em ) {
   Query query = em.createQuery( "select c.no, c.name from Category c" );

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

   for( Object[] row : list ) {

       System.out.println( row[0] + ":" + row[1] );

   }
}
```

5.2.1 기본 문법과 쿼리 API

[3] 여러 값 조회

- 스칼라 타입뿐만 아니라 엔티티 타입도 여러 값을 함께 조회할 수 있다.
- 조회한 엔티티는 영속성 컨텍스트에서 관리된다.

```
public static void testQuery3( EntityManager em ) {

    Query query = em.createQuery( "select b.title, b.category from Book b" );

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

    for( Object[] row : list ) {
        System.out.println( row[0] + ":" + row[1] );
    }
}
```

5.2.1 기본 문법과 쿼리 API

6) New 명령어

testQuery2 에서 no, name 두 필드를 프로젝션해서 타입을 지정할 수 없으므로 TypeQuery 를 사용할 수 없다. 따라서 Object[]을 반환 받았다. 실제 애플리케이션 개발 시에는 Object[] 을 직접 사용하지 않고 다음 코드의 CategoryDTO 처럼 의미 있는 객체로 변환해서 사용하는 것이 보통이다.

```
public static void testQuery2( EntityManager em ) {

   Query query = em.createQuery( "select c.no, c.name from Category c" );
   List<Object[]> resultList = query.getResultList();

   List<CategoryDTO> categoryDTOs = new ArrayList<CategoryDTO>();
   for( Object[] row : resultList ) {
        CategoryDTO categoryDTO = new CategoryDTO();
        categoryDTO.setNo( (Long)row[ 0 ] );
        categoryDTO.setName( (String)row[ 1 ] );

        categoryDTOs.add( categoryDTO );
   }
}
```

위의 방식의 단점은 객체 변환 작업을 직접 해주어야 하기 때문에 지루하고 불편하다.

5.2.1 기본 문법과 쿼리 API

6) New 명령어

new 명령어를 사용하면 TypedQuery를 사용하면서 불편하고 지루한 작업을 조금 줄일 수 있다.

```
public static void testQuery4( EntityManager em ) {

   TypedQuery<CategoryDTO> query =
                em.createQuery( "select com.estsoft.jpabookmall.dto.CategoryDTO( c.no, c.name ) from Category c“, CategoryDTO.class );

   List<CategoryDTO> resultList = query.getResultList();
   for( CategoryDTO result : resultList ) {
      System.out.println( result );
   }
}
```

- 패키지 명을 포함한 전체 클래스 명을 입력해야 한다.

- 순서와 타입이 일치하는 생성자가 필요하다.

5.2.2 페이징 API

7) 페이징 처리

- 페이징 처리용 SQL을 작성하는 일은 지루하고 반복적이다.
- 더 큰 문제는 데이터베이스마다 페이징을 처리하는 SQL 문법이 다르다는 점이다.

JPA는 페이징을 다음 두 API로 추상화했다.

setFirstResult(int startPosition) : 조회 시작 위치 (0부터 시작한다.)
setMaxResults(int maxResult) : 조회할 데이터 수

```
TypedQuery<Member> query =
        em.createQuery( "SELECT m FROM Board b ORDER BY b.regDate DESC", Board.class);

query.setFirstResult( ( page-1) * 10 );
query.setMaxResults( 10 );

List<Board> list = query.getResultList();
```

5.2.3 집합과 정렬

1) 집합

- 집합은 집합함수와 함께 통계 정보를 구할 때 사용한다.

- 다음 코드는 순서대로 회원 수, 나이 합, 평균 나이, 최대 나이, 최소 나이를 조회한다.

```
select

   COUNT(m),   //회원수
   SUM(m.age), //나이 합
   AVG(m.age), //평균 나이
   MAX(m.age), //최대 나이
   MIN(m.age)  //최소 나이

from Member m
```

5.2.3 집합과 정렬

1) 집합

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` , `BigInteger` 합: `BigInteger` , `BigDecimal` 합: `BigDecimal`

- NULL 값은 무시하므로 통계에 잡히지 않는다. (DISTINCT 가 정의되어 있어도 무시된다.)
- 만약 값이 없는데 SUM , AVG , MAX , MIN 함수를 사용하면 NULL 값이 된다. 단 COUNT 는 0이 된다.
- DISTINCT 를 집합 함수 안에 사용해서 중복된 값을 제거하고 나서 집합을 구할 수 있다.

예) select COUNT(DISTINCT u.age) from User u

5.2.3 집합과 정렬

1) 정렬 (Order By)

- ORDER BY 는 결과를 정렬할 때 사용한다.
- 다음은 나이를 기준으로 내림차순으로 정렬하고 나이가 같으면 이름을 기준으로 오름차순으로 정렬한다.

```
select u from User u order by u.age DESC, u.name ASC
```

ASC: 오름차순 (기본값)

DESC: 내림차순

5.2.4 JPQL 조인

JPQL도 조인을 지원하는데 SQL 조인과 기능은 같고 문법만 약간 다르다.

```
SELECT b FROM  Board b ( INNER ) JOIN  b.user  u
```

- 내부 조인 구문을 보면 SQL의 조인과 약간 다른 것을 확인할 수 있다.

- JPQL 조인의 가장 큰 특징은 **연관 필드**를 사용한다는 것이다. 여기서 b.user 가 연관 필드인데 연관 필드는 다른 엔티티와 연관관계를 가지기 위해 사용하는 필드를 말한다.

- FROM Board b : 게시판을 선택하고 b 라는 별칭을 주었다.

- Board b JOIN b.user u : 게시물이 가지고 있는 연관 필드로 User와 조인한다. 조인한 User에는 u 라는 별칭을 주었다.

- JPQL 조인을 SQL 조인처럼 사용하면 문법 오류가 발생한다. JPQL은 JOIN 명령어 다음에 조인할 객체의 **연관 필드**를 사용한다. 다음은 잘못된 예이다.

```
SELECT b FROM  Board b JOIN  User u
```

5.3 QueryDSL

5.3.1 소개

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

2) IDE 자동완성 기능의 도움을 받을 수 있는 등 여러 가지 장점이 있다.

3) Criteria의 가장 큰 단점은 너무 복잡하고 어렵다는 것이다. 작성된 코드를 보면 그 복잡성으로 인해 어떤 JPQL이 생성될지 파악하기가 쉽지 않다.

4) 쿼리를 문자가 아닌 코드로 작성해도, 쉽고 간결하며 그 모양도 쿼리와 비슷하게 개발할 수 있는 프로젝트

5) QueryDSL도 Criteria처럼 JPQL 빌더 역할을 하는데 JPA Criteria를 대체한다.

6) Quer오픈소스 프로젝트로 처음에는 HQL(하이버네이트 쿼리언어)을 코드로 작성할 수 있도록 해주는 프로젝트로 시작해서 지금은 JPA, JDO, JDBC, Lucene, Hibernate Search, MongoDB, 자바 컬렉션등을 다양하게 지원한다.

7) QueryDSL은 이름 그대로 쿼리 즉 데이터를 조회하는데 기능이 특화되어 있다.

5.3.2 설치 및 설정

1) 라이브러리 추가

```
<!-- queryDSL -->
<dependency>
    <groupId>com.mysema.querydsl</groupId>
    <artifactId>querydsl-jpa</artifactId>
    <version>3.6.2</version>
</dependency>

<dependency>
    <groupId>com.mysema.querydsl</groupId>
    <artifactId>querydsl-apt</artifactId>
    <version>3.6.2</version>
    <scope>provided</scope>
</dependency>
```

- querydsl-jpa : QueryDSL JPA 라이브러리

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

5.3.2 설치 및 설정

2) 빌드 환경 설정

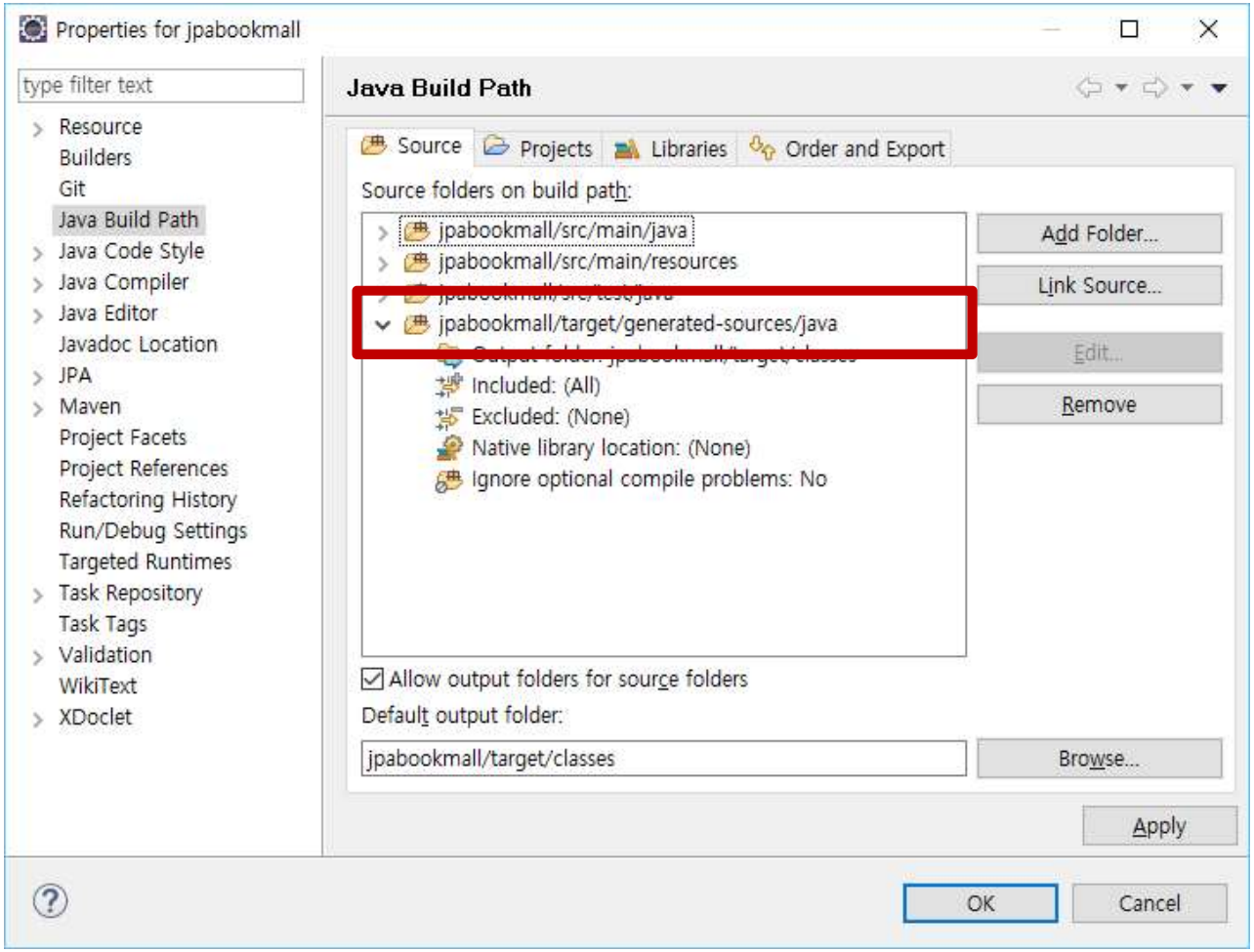
QueryDSL을 사용하려면 Criteria의 엔티티를 기반으로 쿼리 타입이라는 쿼리용 클래스를 생성해야 한다.

```
<build>
   <plugins>
      <plugin>
         <groupId>com.mysema.maven</groupId>
         <artifactId>apt-maven-plugin</artifactId>
         <version>1.0.9</version>
         <executions>
            <execution>
               <goals>
                  <goal>process</goal>
               </goals>
               <configuration>
                  <outputDirectory>target/generated-sources/java</outputDirectory>
                  <processor>com.mysema.query.apt.jpa.JPAAnnotationProcessor</processor>
               </configuration>
            </execution>
         </executions>
      </plugin>
   </plugins>
</build>
```

5.3.2 설치 및 설정

2) 빌드 환경 설정

target/generated-sources/java 디렉토리를 소스경로에 추가 한다.



5.3.3 시작하기

예제 기본 테이터 넣기

```
public static void insertCategories( EntityManager em ) {
// no = 1L
    Category category1 = new Category();
    category1.setName( "Java Programming" );
    em.persist( category1 );

// no = 2L
    Category category2 = new Category();
    category2.setName( "Spring Framework" );
    em.persist( category2 );

// no = 3L
    Category category3 = new Category();
    category3.setName( "C Programming" );
    em.persist( category3 );

}
```

```
public static void insertBooks( EntityManager em ) {

  Category category1 = em.find( Category.class, 1L );
  Book book1 = new Book();
   book1.setTitle( "Effective Java" );
   book1.setPrice( 10000 );
   book1.setCategory( category1 );
   em.persist( book1 );

   Book book2 = new Book();
   book2.setTitle( "Java in a Nutshell" );
   book2.setPrice( 25000 );
   book2.setCategory( category1 );
   em.persist( book2 );

   Book book3 = new Book();
   book3.setTitle( "Java Thread" );
   book3.setPrice( 31000 );
   book3.setCategory( category1 );
   em.persist( book3 );


   Category category2 = em.find( Category.class, 2L );
   Book book4 = new Book();
   book4.setTitle( "Spring in Action" );
   book4.setPrice( 30000 );
   book4.setCategory( category2 );
   em.persist( book4 );

   Book book5 = new Book();
   book5.setTitle( "Spring3 Recipes" );
   book5.setPrice( 27000 );
   book5.setCategory( category2 );
   em.persist( book5 );


  Category category3 = em.find( Category.class, 3L );
   Book book6 = new Book();
   book6.setTitle( "The C Programming Language" );
   book6.setPrice( 30000 );
   book6.setCategory( category3 );
   em.persist( book6 );
```

5.3.3 시작하기

간단한 예제 작성하고 테스트 해보기

```
public static void testQueryDSL( EntityManager em ) {
    JPAQuery query = new JPAQuery(em);
    QBook qBook = new QBook( "b" );            //생성되는 JPQL의 별칭이 m
    List<Book> books =
       query.
       from( qBook ).
       where( qBook.title.eq( "Java" ) ).
       orderBy( qBook.title.desc() ).
       list( qBook );

      for( Book book : books ) {
           System.out.println( book );
      }
}
```

1) QueryDSL을 사용하려면 우선 com.mysema.query.jpa.impl.JPAQuery 객체를 생성해야 한다.

2) 사용할 쿼리 타입(Q)을 생성하는데 생성자에는 별칭이름을 주면 된다. 이 별칭을 JPQL에서 별칭으로 사용한다.

3) 다음에 나오는 from , where , orderBy , list 는 코드만 보아도 쉽게 이해가 될 것이다.

5.3.3 시작하기

4) 기본 Q 생성

- 쿼리 타입(Q) 또는 쿼리 클래스는 사용하기 편리하도록 기본 인스턴스를 보관하고 있다.

- 같은 엔티티를 조인하거나 같은 엔티티를 서브쿼리에 사용하면 같은 별칭이 사용되므로 이때는 별칭을 직접 지정해서 사용해야 한다.

```
public class QBook extends EntityPathBase<Book> {
    public static final QBook book = new QBook( "book" );
        …
    }
```

- 따라서, 별칭이 필요 없는 경우에는 QBook.book 으로 접근하여 사용할 수 있다.

```
JPAQuery query = new JPAQuery( em );
List<Book> list = query.from( QBook.book ).where( QBook.book.title.eq( "Effective Java" ) ).list( QBook.book );

for( Book book : list ) {
    System.out.println( book );
}
```

5.3.4 검색 조건 쿼리

where 절에는 and 나 or 을 사용할 수 있다.

[예제] 책 이름이 "Java Thread" 이고 책 값이 40,000원 이하인 책 검색

```
public static void testSearchCond( EntityManager em ) {
   JPAQuery query = new JPAQuery( em );
   List<Book> list =
               query.from( book ).where( book.title.eq( "Java Thread" ).and( book.price.lt( 40000L ) ) ).list( book );

   for( Book book : list ) {
      System.out.println( book );
   }
}
```

다음과 같이 사용해도 and 연산이 된다.

```
.where( book.title.eq( "Java Thread" ), book.price.lt( 40000L ) ) ;
```

5.3.4 검색 조건 쿼리

쿼리 타입의 필드는 필요한 대부분의 메서드를 명시적으로 제공한다

[실습 과제] 다음 예들을 직접 작성하여 확인해 보자.

```
book.price.between( 10000,20000 );// 가격이 10000원 ~ 20000 원 책

book.title.contains( "Spring“ ); // Spring이라는 이름을 포함한 책

book.title.startsWith( "Java“ ); //이름이 “Java”로 시작하는 상품
```

코드로 작성되어 있으므로 IDE가 제공하는 코드 자동 완성 기능의 도움을 받으면 필요한 메서드를 손 쉽게 찾을 수 있다.

5.3.5 결과조회

1) 쿼리 작성이 끝나고 결과 조회 메서드를 호출하면 실제 데이터베이스를 조회한다.

2) 보통 uniqueResult() 나 list() 를 사용하고 파라미터로 프로젝션 대상을 넘겨준다.

3) 대표적인 결과 조회 메서드는 다음과 같다.

uniqueResult() :
조회 결과가 1 건일 때 사용한다. 조회 결과가 없으면 null 을 반환
조회 결과가 하나 이상이면 com.mysema.query.NonUniqueResultException 예외가 발생

singleResult() :
uniqueResult() 와 같지만 결과가 하나 이상이면 처음 데이터를 반환한다.

list() :
결과가 하나 이상일 때 사용한다. 결과가 없으면 빈 컬렉션을 반환한다.

5.3.6 프로젝션과 결과 반환

1) 프로젝션 대상이 하나인 경우

프로젝션 대상이 하나면 해당 타입으로 반환한다.

```
public static void testProjectionOne( EntityManager em ) {

    JPAQuery query = new JPAQuery( em );
    List<String> list = query.from( book ).list( book.title );

    for( String title : list ) {
        System.out.println( title );
    }
}
```

5.3.6 프로젝션과 결과 반환

2) 여러 컬럼 반환과 튜플(Tuple)

프로젝션 대상으로 여러 필드를 선택하면 QueryDSL은 기본으로 com.mysema.query.Tuple 이라는 Map 과 비슷한 내부 타입을 사용한다. 조회 결과는 tuple.get() 메서드에 조회한 쿼리 타입을 지정하면 된다.

```
public static void testProjectionTuple( EntityManager em ) {

    JPAQuery query = new JPAQuery( em );
    List<Tuple> list = query.from( book ).list( book.title, book.price );

    for( Tuple tuple : list ) {
        System.out.println( tuple.get( book.title ) + ":" + tuple.get( book.price ) );
    }
}
```

5.3.6 프로젝션과 결과 반환

3) 빈 생성(Bean Population)

- 쿼리 결과를 엔티티가 아닌 특정 객체로 받고 싶으면 **빈 생성 기능**을 사용
- 원하는 방법을 지정하기 위해 com.mysema.query.types.Projections 를 사용
- **다양한 빈 생성 기능**

  프로퍼티 접근

  필드 직접 접근

  생성자 사용

5.3.6 프로젝션과 결과 반환

3) 빈 생성(Bean Population)

프로퍼티 접근

```
public static void testProjectionBean( EntityManager em ) {

   JPAQuery query = new JPAQuery( em );
   List<CategoryDTO> list =
           query.from( category ).list( Projections.bean( CategoryDTO.class, category.no.as( "no" ), category.name ) );

   for( CategoryDTO dto : list ) {
       System.out.println( dto );
   }
}
```

- Projections.bean() 메서드는 수정자(Setter)를 사용해서 값을 채운다.

- 쿼리 결과와 매핑할 프로퍼티 이름이 다르면 as 를 사용해서 별칭을 주면 된다.

5.3.6 프로젝션과 결과 반환

3) 빈 생성(Bean Population)

필드 접근

```
public static void testProjectionBean( EntityManager em ) {

   JPAQuery query = new JPAQuery( em );
   List<CategoryDTO> list =
           query.from( category ).list( Projections.fields ( CategoryDTO.class, category.no.as( "no" ), category.name ) );

   for( CategoryDTO dto : list ) {
       System.out.println( dto );
   }
}
```

- Projections.fields() 메서드를 사용하면 필드에 직접 접근해서 값을 채워준다.(setter가 없어도)

- 필드를 private 으로 설정해도 값을 채운다.

5.3.6 프로젝션과 결과 반환

3) 빈 생성(Bean Population)

생성자 접근

```
public static void testProjectionBean( EntityManager em ) {

   JPAQuery query = new JPAQuery( em );
   List<CategoryDTO> list =
           query.from( category ).list( Projections.constructor( CategoryDTO.class, category.no.as( "no" ), category.name ) );

   for( CategoryDTO dto : list ) {
       System.out.println( dto );
   }
}
```

- Projections.constructor() 메서드는 생성자를 사용할 수 있다.

- 지정한 프로젝션과 파라미터 순서가 같은 생성자가 필요하다.

5.3.7 페이징

페이징은 offset 과 limit 을 적절히 조합해서 사용하면 된다.

```
public static void testPaging( EntityManager em ) {

   JPAQuery query = new JPAQuery( em );
   List<Book> list = query.from( book ).offset(3).limit(3).list( book );

   for( Book book : list ) {
      System.out.println( book );
   }
}
```

보통, list() 대신에 listResults() 를 사용하게 된다. 이유는 실제 페이징 처리를 하려면 검색된 전체 데이터 수를 알아낼 때, 유용하기 때문이다.

5.3.7 페이징

페이징은 offset 과 limit 을 적절히 조합해서 사용하면 된다.

```
public static void testPaging2( EntityManager em ) {
    JPAQuery query = new JPAQuery( em );
    SearchResults<Book> results = query.from( book ).offset(3).limit(3).listResults( book );

l   ong total = results.getTotal(); //검색된 전체 데이터 수
    long limit = results.getLimit();
    long offset = results.getOffset();

    System.out.println( total + ":" + limit + ":" + offset );
    for( Book book : results.getResults() ) {
        System.out.println( book );
   }
}
```

보통, list() 대신에 listResults() 를 사용하게 된다. 이유는 실제 페이징 처리를 하려면 검색된 전체 데이터 수를 알아낼 때, 유용하기 때문이다.

5.3.8 정렬

정렬은 orderBy 를 사용하는데 쿼리타입(Q)이 제공하는 asc() , desc() 를 사용

```
.orderBy ( book.price.desc(), book.title.asc() )
```

5.3.9 조인(Join)

- 조인은 innerJoin(join) , leftJoin , rightJoin , fullJoin을 사용
- 추가로 JPQL의 on 과 성능 최적화를 위한 fetch 조인도 사용 가능

> join(조인 대상, 별칭으로 사용할 쿼리 타입)

1) 기본 조인

```
public static void testJoin( EntityManager em ) {

   JPAQuery query = new JPAQuery( em );
   List<Book> list = query.from( book ).join( book.category, category ).list( book );

   for( Book book : list ) {
      System.out.println( book );
   }
}
```

5.3.9 조인(Join)

2) on 사용

```
public static void testJoin2( EntityManager em ) {

    JPAQuery query = new JPAQuery( em );
    List<Book> list =
            query.from( book ).join( book.category, category ).on( category.name.like( "%J%" ) ).list( book );

    for( Book book : list ) {
        System.out.println( book );
    }
}
```

3) from 절에 여러 조건 사용

```
public static void testJoin3( EntityManager em ) {
    JPAQuery query = new JPAQuery( em );
    List<Book> list = query.from( book, category ).where( book.category.eq( category) ).list( book );

    for( Book book : list ) {
        System.out.println( book );
    }
}
```

5.3 QueryDSL

5.3.8 삭제, 수정

1) QueryDSL도 수정, 삭제 같은 배치 쿼리를 지원한다.

2) JPQL 배치 쿼리와 같이 영속성 컨텍스트를 무시하고 데이터베이스를 직접 쿼리 한다는 점에 유의

3) 수정

```
public static void testUpdate( EntityManager em ) {
   JPAUpdateClause updateClause = new JPAUpdateClause( em, book );
   long count =
          updateClause.
          where( book.price.gt( 20000 ) ).
          set( book.price,    book.price.subtract( book.price.multiply( 20 / 100 ) ) ).
         execute();

   System.out.println( "Update: " + count );
}
```

5.3.8 삭제, 수정

4) 삭제

```
public static void testDelete( EntityManager em ) {

    JPADeleteClause deleteClause = new JPADeleteClause( em, book );

    long count = deleteClause.where( book.title.like( "%Java%" ) ).execute();

   System.out.println( "Delete: " + count );
}
```