



ORM

OBJECT RELATIONAL MAPPING

JDBC

- JDBC API
(JAVA)

SQL Mapper

- MyBatis
- (Spring)
JDBC

ORM

- JPA
- (Spring)
JDBC

JDBC (Java Database Connectivity)

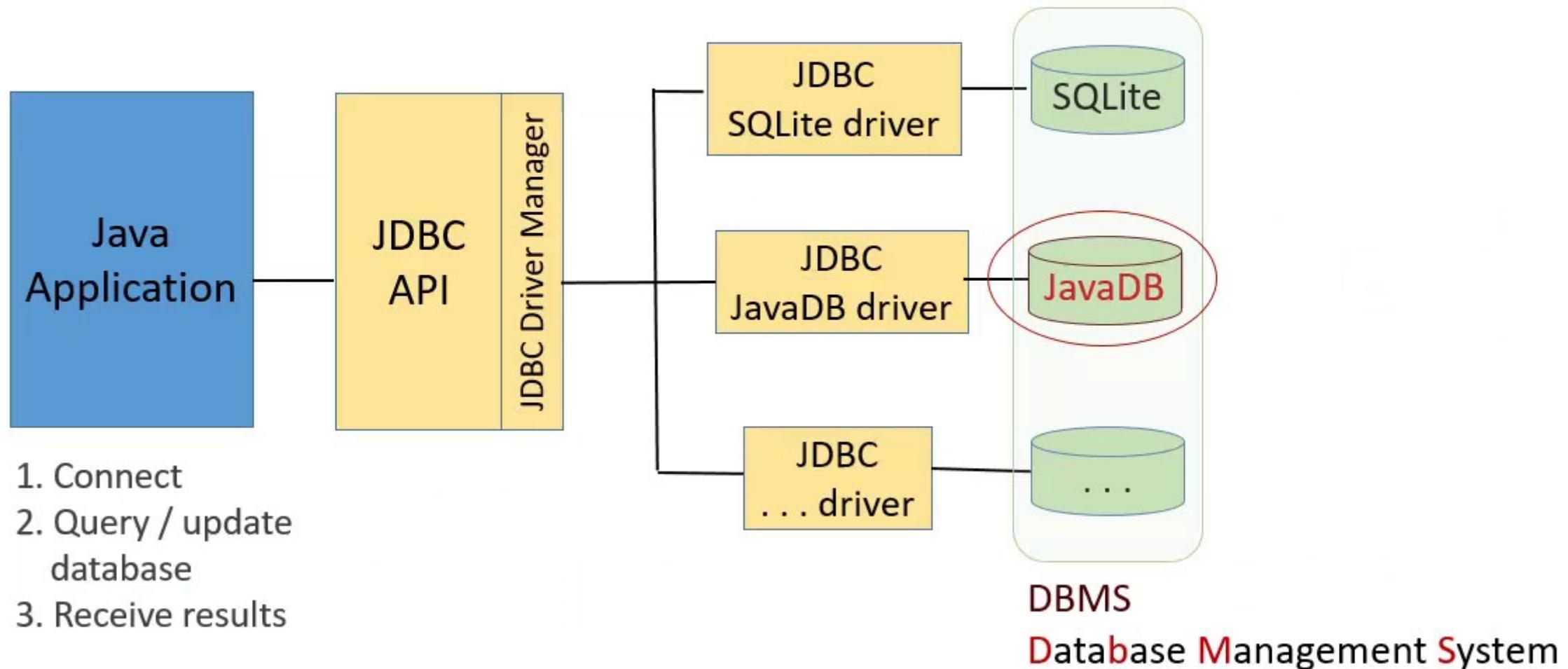


JDBC? Java 진영의 Database 연결 표준
인터페이스



등장 배경: 온라인 비즈니스 투자 증가 ->
DB Connector에 대한 수요

JDBC - Java Database Connectivity



JDBC의 단점 ?

- JDBC 드라이버를 로딩
- 데이터베이스와 연결
- Connection 객체 관리
 - > 이 모든 것을 일일이, 반복적으로 해줘야 함
- SQL문이 자바 코드 내부에 String 형태로 삽입되어 있음.

```
public int insert(Board board) {  
    String url = "jdbc:oracle:thin:@localhost:1521:xe";  
    String user = "kosta211";  
    String password = "1234";  
    String sql = "insert into board values(board_seq.nextval, ?, ?, ?, sysdate, 0)"  
    int re = -1;  
    Connection conn = null;  
    PreparedStatement pstmt = null;  
  
    try {  
        //1. JDBC 드라이버를 로딩  
        Class.forName("oracle.jdbc.driver.OracleDriver");  
        //2. 데이터베이스와 연결 (Connection 객체 생성)  
        conn = DriverManager.getConnection(url, user, password);  
        //3. SQL 질의 (PreparedStatement 객체 생성)  
        pstmt = conn.prepareStatement(sql);  
        pstmt.setString(1, board.getTitle());  
        pstmt.setString(2, board.getWriter());  
        pstmt.setString(3, board.getContents());  
  
        re = pstmt.executeUpdate();  
  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally {  
        if(pstmt != null) { // 메모리 낭비 최소화  
            try {  
                pstmt.close();  
            } catch (Exception e2) {}  
        }  
  
        if(conn != null) {  
            try {  
                conn.close();  
            } catch (Exception e3) {}  
        }  
    }  
    return re;  
}
```

SQL Mapper – MyBatis

- SQL과 Java 코드 분리에 초점

1. Configuration 분리

: SQL 관련 configuration을 별도의 xml 파일로 분리

2. SQL문 분리

: SQL문을 xml tag에 넣어주고, 이 xml tag 하나가
자바 메서드 하나로 mapping

+ 쿼리 실행 결과도 자바 클래스 하나로 mapping

```
<insert id="insertBoard" parameterType="Board">
    insert into board(
        SEQ, TITLE, WRITER, CONTENTS, REGDATE, HITCOUNT
    )
    values(
        BOARD_SEQ.NEXTVAL, #{title}, #{writer}, #{contents}, SYSDATE, 0
    )
</insert>
```

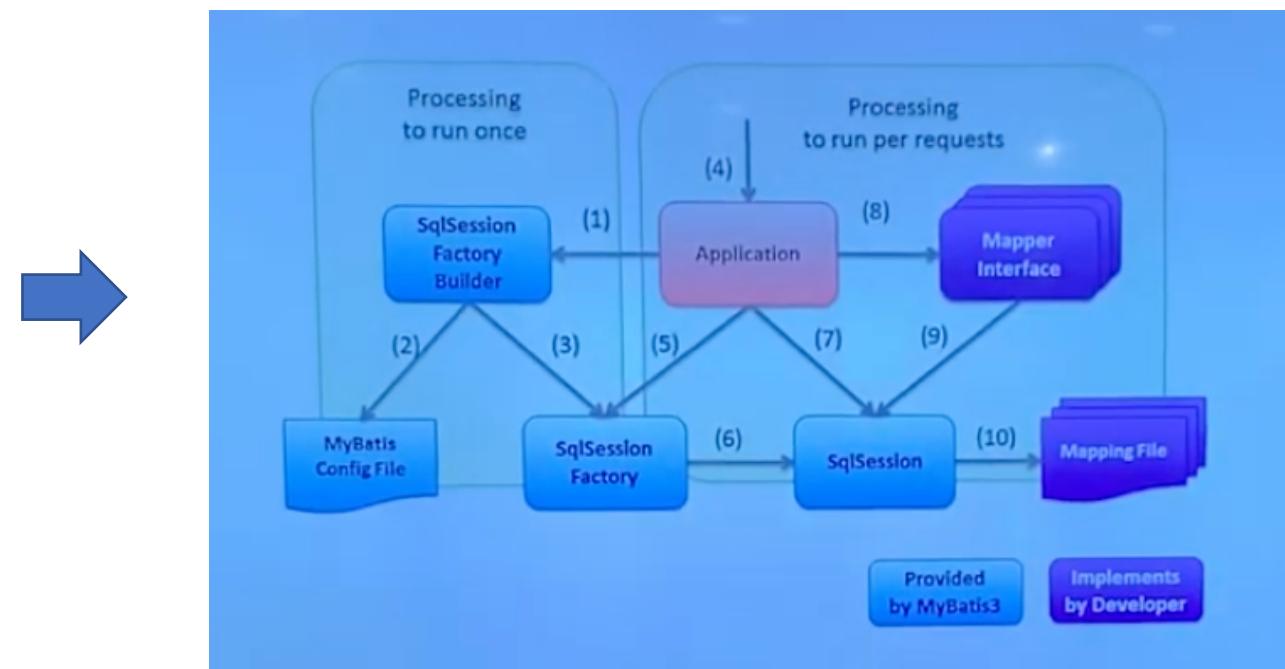
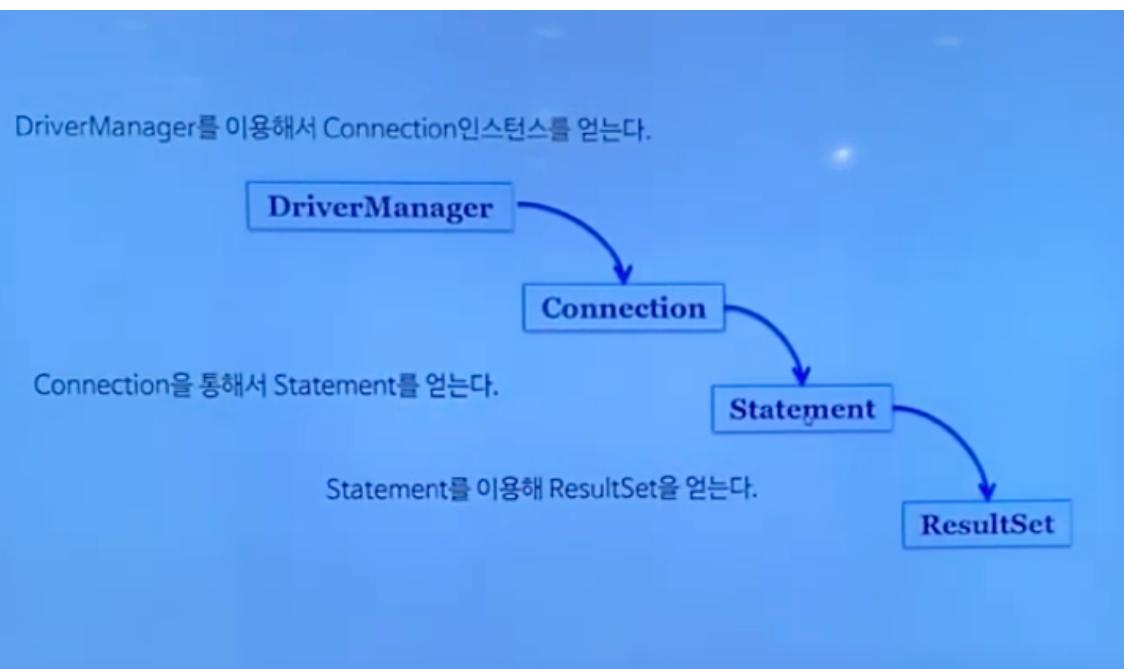
<map statement>

SQL Mapper – MyBatis

- 동작 과정
 - xml 파일을 읽어서 SqlSessionFactory 생성
 - SqlSession 생성 (xml에 있는 여러 SQL과 맵핑된 메서드들을 사용하기 위해 xml 파일을 객체화한 것)
- 장점
 - SQL과 Java code 분리
 - SQL 질의 결과로 얻은 ROW를 자바 객체로 mapping해줌
 - configuration의 분리로 DAO(Data Access Object)가 간단해짐

```
public SqlSessionFactory getSqlSessionFactory() {  
    String resource = "mybatis-config.xml";  
    InputStream in = null; // reader도 가능  
  
    try {  
        //mybatis-config.xml에 input stream을 연결  
        in = Resources.getResourceAsStream(resource);  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
  
    return new SqlSessionFactoryBuilder().build(in);  
}  
  
public int insertBoard(Board board) {  
    int re = -1;  
  
    SqlSession sqlSession = getSqlSessionFactory().openSession();  
    try {  
        re = sqlSession.getMapper(BoardMapper.class).insertBoard(board);  
        if (re > 0) {  
            sqlSession.commit(); // mybatis만 쓰면 이렇게 트랜잭션 처리해줘야함  
        } else {  
            sqlSession.rollback();  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally {  
        if (sqlSession != null) {  
            sqlSession.close();  
        }  
    }  
    return re;  
}
```

SQL Mapper – MyBatis



ORM(OBJECT RELATIONAL MAPPING)

객체지향을 객체지향답게

이전까지 DB Connector들의 문제 ?



객체지향 프로그램과 관계형 데이터베이스
사이의 Mapping이 어려움 !

객체지향 VS 관계형 데이터베이스

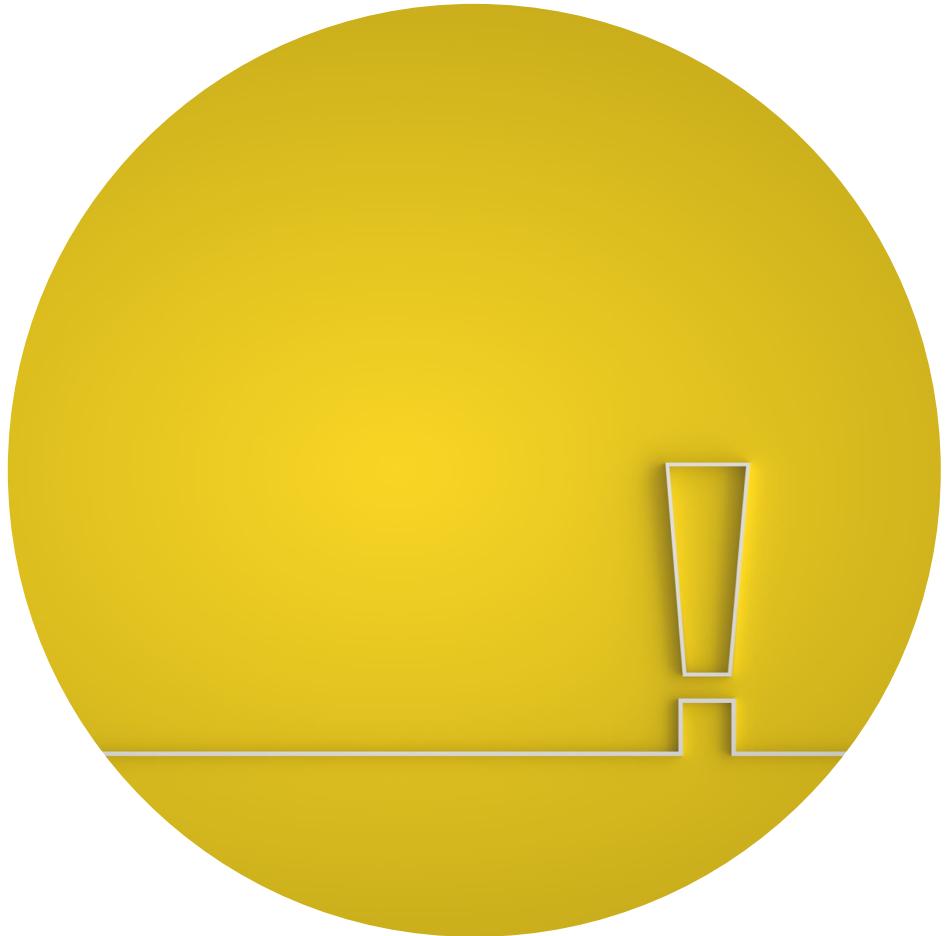
- Post 클래스 안에 Author 객체를 property로 가지고 있는 구조
- post table에서는 author 객체 대신 author_id를 가짐
- + comments에서 post_id를 가짐

```
public class Post {  
    private int id;  
    private List<Comment> comments;  
    private Author author;  
    private String title;  
    private String date;  
    private String content;
```

컬럼명	#	Type	Type Mod	Not Null	디폴트	Comment
123 ID	1	NUMBER		[X]		
ABC TITLE	2	VARCHAR2(50)		[]		
⌚ PUBLISHED	3	DATE		[]		
ABC CONTENT	4	VARCHAR2(200)		[]		
123 AUTHOR_ID	5	NUMBER		[]		

Entity Manager

- 엔티티의 저장, 수정, 삭제, 조회 등 엔티티와 관련된 모든 일을 처리하는 관리자 (find, persist, merge, remove 등 메소드를 가지고 있음)

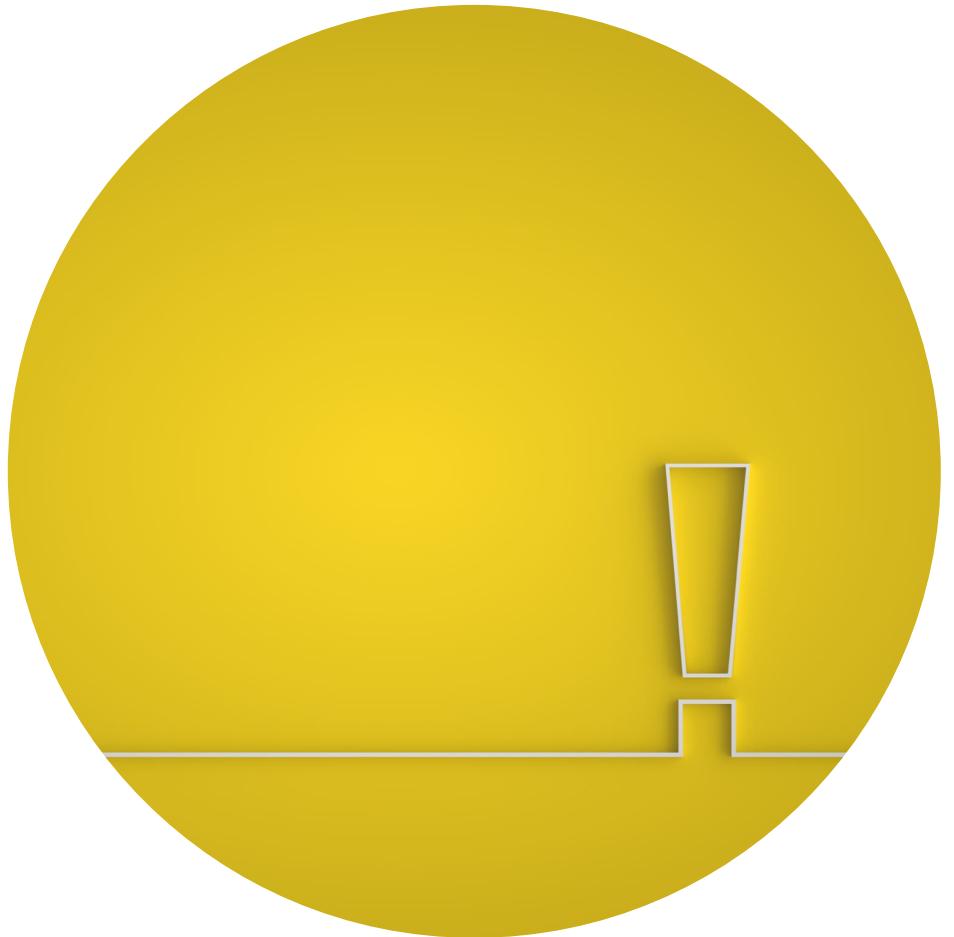


Entity

- JPA를 통해서 데이터베이스의 테이블과 mapping할 클래스

어노테이션

- `@Entity`: JPA가 관리할 객체임을 명시
- `@Table`: 맵핑할 데이터베이스 테이블 이름을 명시
- `@Id`: 기본 키(PK)로 맵핑할 속성
- `@Column`: 필드(attribute)와 테이블의 컬럼 맵핑



예) Entity 설정

- ItemEntity 클래스

```
@Entity                    -> JPA가 관리할 객체
@Table(name = "Items")      -> 매핑할 테이블 이름
public class ItemEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "item_id")
    private Long itemId;

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

    private Long price;
}
```

기본 키 매핑

필드와 칼럼 매핑

엔티티 간의 관계

- 관계형 데이터베이스와 SQL Mapper: 기본적으로 양방향 관계
(조인을 하면 어느 쪽 엔티티에서든 자신과 관계를 맺고 있는
엔티티를 참조 가능)
- JPA: 양방향, 단방향 관계 중 선택 가능

방향성

- 단방향 : 두 엔티티가 관계를 맺을 때, 한 쪽의 엔티티만 참조하고 있는 것
- 양방향 : 두 엔티티가 관계를 맺을 때, 양 쪽이 서로 참조하고 있는 것

양방향 연관 관계

- 관계의 주인(owner)을 설정해줘야 함
- 연관 관계의 주인은 외래 키가 있는 곳(1:N에서는 N쪽이 주인)
- 연관관계의 주인만이 외래 키를 관리(등록, 수정, 삭제) 할 수 있고, 반면 주인이 아닌 엔티티는 읽기만 할 수 있음
- 연관 관계의 주인이 아닌 경우, mappedBy 속성으로 연관 관계의 주인을 지정해줄 수 있음

외래 키(FK) 맵핑

```
@Entity  
@Table( name="book")  
public class Book {  
  
    @Id  
    @Column(name="no")  
    @GeneratedValue( strategy = GenerationType.IDENTITY )  
    private Integer no;  
  
    @Column(name="title", nullable=false, length=200)  
    private String title;  
  
    @ManyToOne  
    @JoinColumn(name = "category_no")  
    private Category category;  
  
    // getter , setter 생략  
  
}
```

=> 단방향 관계: Book에서 Category 정보들을 가져올 수 있음

- `@JoinColumn` : 외래키로 실제 테이블의 어떤 컬럼을 설정해줄지 정해줌
- 다중성(Multiplicity)
- `@OnetoOne`
- `@OneToMany`
- `@ManyToOne`
- `@ManyToMany`

영속성 전이

- CascadeType.ALL
 - CascadeType.PERSIST
 - CascadeType.MERGE
 - CascadeType.REMOVE
 - CascadeType.REFRESH
 - CascadeType.DETACH
- 엔티티가 로딩될 때
연관된 엔티티가 같이
로딩되도록 하는 것: Eager
- 실제로 연관관계 있는
엔티티를 참조할 때 로딩:
Lazy Loading

페치 전략(Fetch Strategy)

- FetchType.EAGER
- FetchType.LAZY

영속성 컨텍스트(Persistence Context)

- EntityManager는 데이터를 바로 DB에 저장하는 것이 아니라, 영속성 컨텍스트에 저장하는 중간 단계를 거침

영속성(Persistence)

- 실행 중인 프로그램에서 만들어진 데이터를 실제 DB에 저장하는 것

엔티티의 생명주기

- 비영속(new/transient)

: 객체를 생성만 한 상태

- 영속(managed)

: 영속성 컨텍스트에 저장된 상태(관리되고 있는 상태)

- 준영속(detached)

: 영속성 컨텍스트에 저장했다가 지운 상태

- 삭제(removed)

: 실제 DB에서 삭제를 요청한 상태

플러시(flush)

- 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영
- 영속성 컨텍스트에서 엔티티를 지우는 것이 아니라, 변경 내용을 DB에 동기화하는 것

- **플러시의 흐름**

1. 변경 감지가 동작해서 스냅샷과 비교해서 수정된 엔티티를 찾는다.
2. 수정된 엔티티에 대해서 수정 쿼리를 만들고 SQL 저장소에 등록한다.
3. 쓰기 지연 SQL 저장소의 쿼리를 데이터베이스에 전송한다.

- **플러시하는 방법**

1. em.flush()
2. 트랙잭션 커밋시 자동 호출
3. JPQL 쿼리 실행시 자동 호출