JPA Association

558280-1
2026년 봄학기
5/7/2026
박경신

왜 JPA Association이 어려운가?

- 객체 모델과 DB 모델이 근본적으로 다름
  - 객체 모델 : 참조(reference) 기반 그래프
    - member.getOrders().get(0).getItem();
  - DB 모델 : 외래키(FK) 기반 그래프
    - SELECT * FROM member m **JOIN orders o ON m.id = o.member_id**
- 문제 -> 단순 문법이 아니라 "설계 문제"
  - 객체 그래프 vs 테이블 조인 불일치
    - 객체에서는 자연스러운 탐색이 DB에서는 비용이 큰 연산
  - 성능 (N+1) 문제
    - 자바에서 member.getOrders()는 SELECT * FROM member → 1번
      SELECT * FROM orders WHERE member_id=? → N번 (총 N+1 쿼리)
  - 무한 루프 (JSON 직렬화)
    - 직렬화시 Member → Order → Member → Order → ...
  - 연관관계 주인 (owner) 개념
    - 객체는 양방향인데, DB는 FK 하나인데, 누가 관리?

JPA Association Mapping

- 연관 관계(association)는 **테이블과 테이블 (또는 객체와 객체)가 서로 참조하는 관계**를 의미함

주요 키워드	설명
방향(direction)	**단방향(Unidirectional), 양방향(Bidirectional).** 객체가 참조변수를 통해 다른 객체를 참조하면 단방향이고, 만일 다른 객체도 첫 번째 객체를 참조한다면 양방향임.
다수성(multiplicity)	**다대일 (N:1), 일대다 (1:N), 일대일 (1:1), 다대다 (N:N)**
연관관계 주인(owner)	객체를 양방향 연관관계로 만들기 위해서는 연관관계 **주인(owner)**을 정해야 함. 연관관계의 **주인은 외래키(foreign key)를 관리(등록,수정, 삭제 등)** 할 수 있으며, 주인이 아닌 쪽은 읽기만 가능함. 일반적으로, N:1과 1:N 관계에서 연관관계 주인은 N에 해당하는 객체임.

JPA Association Mapping

- Entity들은 서로 다양한 연관 관계를 맺을 수 있음

연관관계	JPA Annotation
1:1	@OneToOne
1:N	@OneToMany
N:1	@ManyToOne
N:N	@ManyToMany

Unidirectional Association

- 단방향 연관(Unidirectional association)은 Entity 간의 관계를 설정하는 데 가장 기본적이고 일반적으로 사용됨
- 하지만 단방향 연관에서는 한 Entity 에서만 다른 Entity 에 대한 참조를 보유한다는 점에 유의해야 함
- 단방향 연관 설정

관계 유형	설정	외래키(FK) 위치
N:1	@ManyToOne + @JoinColumn	연관관계의 주인 (N쪽)
1:N	@OneToMany + @JoinColumn	참조대상 (N쪽)
1:1	@OneToOne + @JoinColumn	주 테이블
N:N	@ManyToMany + @JoinTable	별도의 연결 테이블 생성

Bidirectional Association

- 양방향 연관(Bidirectional association)은 양쪽 Entity 모두가 서로를 참조하는 관계
- 하지만 양방향 연관에서는 연관 관계의 주인(owner)와 비주인(inverse)를 명확히 설정해야 하며, 주인 쪽만이 DB의 외래 키 (Foreign Key, FK)를 관리함
- 양방향 연관은 다음과 같이 구성

관계 유형	주인(Owner) 설정	비주인(Inverse) 설정
N:1/1:N	@ManyToOne (언제나 주인)	@OneToMany(mappedBy = "...")
1:1	@OneToOne + @JoinColumn	@OneToOne(mappedBy = "...")
N:N	@ManyToMany + @JoinTable	@ManyToMany(mappedBy = "...")

Unidirectional vs Bidirectional

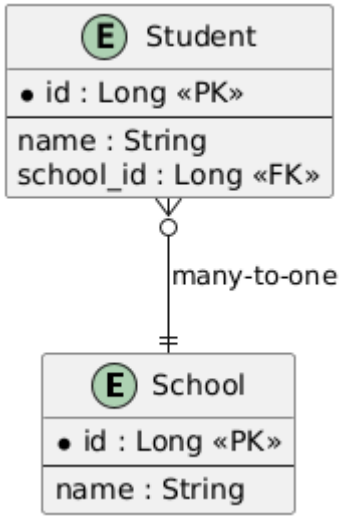
- 단방향과 양방향 연관 관계 비교
  - 양방향 연관은 편리하지만, **순환 참조나 무한 루프(예: toString() 에서 양쪽 호출 등)의 위험**이 있으므로, JSON 직렬화 시 주의 필요 (@JsonManagedReference, @JsonBackReference 또는 DTO 활용).

항목	단방향 연관	양방향 연관
참조 방향	한쪽만 참조	양쪽 모두 참조
탐색성	한 방향 탐색만 가능	양쪽 방향 모두 탐색 가능
외래키 관리	참조하는 쪽에서 관리	주인 쪽에서만 외래키 관리
복잡성	상대적으로 단순	mappedBy 설정 필요

Unidirectional @ManyToOne

- 단방향 @ManyToOne 관계에서 Entity 의 여러 instance는 다른 Entity 의 한 instance 와 연결됨
  - 예로, 학생과 학교 간의 관계 - 각 학생은 하나의 학교에만 등록하나 각 학교는 여러 명의 학생이 있을 수 있음

```
@Entity
public class Student {
    @Id    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @ManyToOne
    @JoinColumn(name = "school_id")
    private School school;
}
@Entity
public class School {
    @Id    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
}
```



Unidirectional @ManyToOne

- N:1 Student -> School 관계
  - "여러 Student가 하나의 School에 속한다"는 다대일 관계
  - 단방향(Unidirectional)이란, Student Entity만 School을 참조하고, School Entity는 Student를 알지 못하는 구조를 의미함
  - DB 상에서는 student 테이블에 외래키(Foreign Key, FK) 컬럼(school_id)이 생성됨
  - 소유자(owner)는 외래키를 가진 Student
  - @JoinColumn(name = "school_id") 명시적 지정
  - 기본 fetch 전략 Fetch.EAGER (즉시 로딩)인데, 이로 인해 불필요한 데이터가 로드될 수 있으므로, Fetch.LAZY 로딩 변경을 권장함
    - Fetch.LAZY는 학교 정보가 필요할 때만 SQL 실행 (성능 최적화)

Unidirectional @ManyToOne

- N:1 Student -> School 관계는 JPA/Hibernate에 의해 다음과 같은 테이블이 생성됨
  - Student 테이블이 School 테이블을 참조하는 외래키를 가짐 (school_id)

```
CREATE TABLE school (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255)
);

CREATE TABLE student (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    school_id BIGINT,
    CONSTRAINT fk_student_school FOREIGN KEY (school_id)
REFERENCES school(id)
);
```

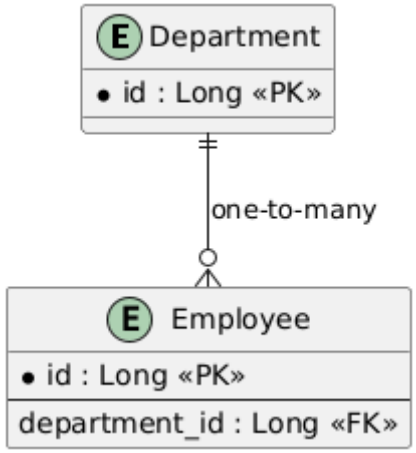
@JoinColumn

- @JoinColumn은 **외래 키(Foreign Key)**를 매핑할 때 사용
- 기본적으로 @Column 이 가지고 있는 unique, nullable, insertable, updatable, columnDefinition, table 속성을 가지고 있으며, 추가로 name, referencedColumnName, foreignKey 와 같은 속성을 가지고 있음
  - name – 매핑할 외래키의 이름을 지정함
  - referencedColumnName – 외래 키가 참조하는 대상 테이블의 컬럼명을 의미하며 기본 값은 참조하는 테이블이 기본 키 컬럼명으로 설정됨
  - foreignKey(DDI) – 외래키 제약조건을 직접 지정할 수 있음. 해당 속성은 테이블을 생성할 때만 사용함
- @JoinColumn name의 기본값은 default로 설정됨
  - **참조하는 Entity 필드명 + “ _” + 참조된 Entity 의 기본 키 명 (id)**

Unidirectional @OneToMany

- 단방향 @OneToMany 관계에서 Entity는 다른 Entity 의 하나 이상의 instance 에 대한 참조를 가짐
  - 예로, 부서와 직원 간의 관계- 각 부서에는 여러 직원이 있지만, 각 직원은 하나의 부서에만 속함

```
@Entity
public class Department {
    @Id
    private Long id;
    @OneToMany
    @JoinColumn(name = "department_id")
    private List<Employee> employees;
}
@Entity
public class Employee {
    @Id
    private Long id;
}
```



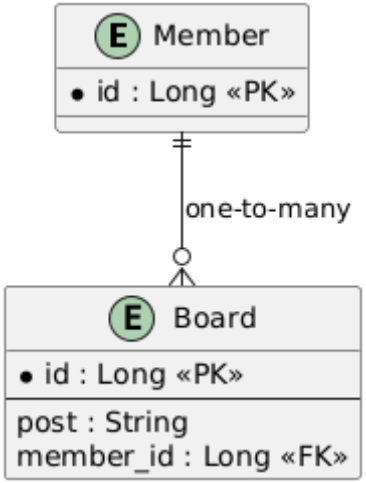
Unidirectional @OneToMany

- 1:N Department -> Employee 관계
  - "하나 Department가 여러 Employee를 가질 수 있다"는 일대다 관계
  - 단방향이란, Department Entity만 Employee 목록을 참조하고, Employee Entity는 Department를 알지 못하는 구조를 의미함
  - @JoinColumn(name = "department_id") 명시적 지정했기 때문에, Employee 테이블에 department_id라는 외래키(FK) 컬럼이 생성됨
  - **외래키는 Employee 테이블에 존재**하지만, JPA 매핑 상에서는 Department가 관계를 관리함
  - **소유자(owner)는 Department (@JoinColumn을 선언했기 때문)**
  - 기본 fetch 전략 Fetch.LAZY
  - @OneToMany 단방향 매핑은 엔티티 모델을 단순하게 유지할 수 있으나, 외래 키 관리 및 성능 이슈에 주의해야 함
  - 대체로 양방향(@OneToMany(mappedBy) + @ManyToOne) 매핑을 사용하여 관계를 명확히 관리하는 방법이 더 일반적임

Bidirectional @OneToMany @ManyToOne

- 이 관계에서는 한 Entity는 다른 Entity에 대한 참조를 가지며, 다른 Entity는 첫 번째 Entity에 대한 참조 컬랙션을 가짐
  - 예로, Member는 Board Entity 컬렉션을 가지며, Board은 자신이 속한 Member에 대한 참조를 가짐

```
@Entity
public class Member {
    @Id @GeneratedValue
    private Long id;
    @OneToMany(mappedBy = "member") // 읽기 전용
    private List<Board> boards;
}
@Entity
public class Board {
   @Id @GeneratedValue
    private Long id;
    private String post;
    @ManyToOne
    @JoinColumn(name = "member_id") // 외래키 (연관관계 주인)
    private Member member;
}
```



Bidirectional @OneToMany @ManyToOne

- 양방향 @OneToMany/@ManyToOne 관계는 JPA에서 가장 일반적이며, 데이터 탐색 및 관계 모델링 정확성을 높임
- @ManyToOne 연관관계의 주인(owner)
  - @JoinColumn(name = "member_id")로 외래키 컬럼 이름 지정
  - **Board가 외래키를 직접 관리하는 주인**
  - Board.member 필드는 insert/update 시 DB에 반영됨
  - 연관관계의 주인(Board.member)에서만 연관관계 저장 가능
- @OneToMany(mappedBy = "member") 역방향(inverse) 관계
  - Member.boards는 외래키를 직접 관리하지 않음
  - mappedBy = "member"는 Board.member 필드를 참조하여 연관관계 주인이 아님을 명시함
  - 단지 Board 컬렉션을 조회하거나 탐색하는 용도임

Bidirectional @OneToMany @ManyToOne

- 이 관계에서 외래키를 가지는 테이블은 Board
  - 외래키 member_id는 Board 테이블에 위치함
  - Member 테이블은 외래키를 갖지 않음

```
CREATE TABLE member (
    id BIGINT PRIMARY KEY AUTO_INCREMENT
);

CREATE TABLE board (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    post VARCHAR(255),
    member_id BIGINT,
    FOREIGN KEY (member_id) REFERENCES member(id)
);
```

mappedBy

- 연관관계의 주인(ownership)
  - JPA에서 양방향 연관관계를 사용할 때 외래 키(Foreign Key)가 있는 곳이 연관관계의 주인임
  - 이 주인 쪽만이 **연관관계에 영향을 주는 insert/update를 수행**할 수 있음 (즉, 데이터베이스에 외래 키를 실질적으로 반영하는 쪽임)
  - **mappedBy는 주인이 아닌 쪽에 설정하여 "나는 읽기만 할 수 있음"을 선언함**
- 단방향 연관관계에서의 주인
  - 단방향의 경우, 연관관계는 한쪽에만 존재하므로 어노테이션(@ManyToOne과 @JoinColumn, 등)이 있는 Entity가 연관관계의 주인
  - 단방향에서는 반대쪽(inverse) 개념이 없으므로, mappedBy 필요없으며 사용할 수도 없음

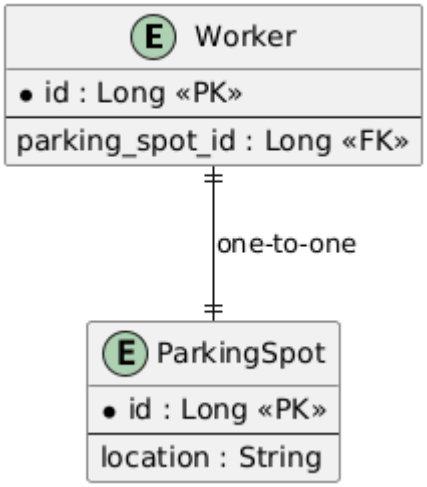
mappedBy

- 양방향 연관관계에서의 주인
  - 양방향 관계는 두 Entity가 서로를 참조
  - mappedBy 설정이 빠지면 DB에 양쪽 모두 외래 키를 생성함 -> 데이터 무결성 문제 발생
- 연관 관계 주인 지정의 원칙
  - 외래키(Foreign Key, FK)를 갖고 있는 쪽이 연관관계의 주인
  - mappedBy는 주인이 아닌 쪽에 지정하여 양방향 완성
  - mappedBy = “ownerField” 형태로 사용하여 **내가 주인이 아님**을 JPA에 알려주는 역할

Unidirectional @OneToOne

- 단방향 @OneToOne 관계에서 한 Entity의 instance 는 다른 Entity 의 한 instance 와만 연결됨
  - 예로, 직원과 주차 공간 간의 관계 - 각 직원은 주차 공간을 가지고 있으며, 각 주차공간은 한 명의 직원에게 속함

```
@Entity
public class Worker {
    @Id
    private Long id;
    @OneToOne
    @JoinColumn(name = "parking_spot_id")
    private ParkingSpot parkingSpot;
}
@Entity
public class ParkingSpot {
    @Id
    private Long id;
    private String location;
}
```



Unidirectional @OneToOne

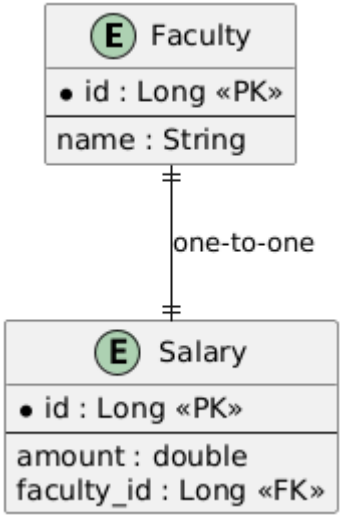
- 1:1 Worker -> ParkingSpot 관계
  - Worker가 ParkingSpot을 참조하지만, ParkingSpot은 Worker를 참조하지 않는 단방향 관계를 나타냄
  - 각 Worker는 하나의 ParkingSpot만 갖고, 해당 ParkingSpot은 오직 하나의 Worker에게만 할당됨
  - **소유자(owner)는 Worker** (외래키 parking_spot_id를 가짐)
  - Worker에서 ParkingSpot을 쉽게 조회 가능 (Fetch Join, EntityGraph 등 활용 가능)
  - 기본 fetch 전략 Fetch.EAGER (즉, Worker를 조회하면 자동으로 ParkingSpot도 함께 로딩됨)

Bidirectional @OneToOne

- 양방향 일대일 관계
  - 일대일 [1:1] 관계는 그 반대도 일대일 관계임
  - 양방향 일대일 관계는 외래 키(Foreign Key: FK)를 넣을 곳을 주인 테이블 또는 대상 테이블 중에 선택 가능함
    - 주인 테이블에 외래 키 저장
    - 대상 테이블에 외래 키 저장
  - 외래 키에 데이터베이스 유니크 제약조건을 추가해야 일대일 관계가 됨

Bidirectional @OneToOne

```
@Entity
@Table(name = "faculty")
public class Faculty {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    // bidirectional one-to-one relationship
    @OneToOne(mappedBy = "faculty", cascade = CascadeType.ALL,
orphanRemoval = true, fetch=FetchType.LAZY)
    private Salary salary;
// getter & setter
}
```



Bidirectional @OneToOne

```
@Entity
@Table(name = "salary")
public class Salary {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private double amount;
    //what column in Salary table has the FK
    @JoinColumn(name = "faculty_id") // 외래키는 salary 테이블에 존재
    @OneToOne(fetch = FetchType.LAZY)
    private Faculty faculty;
    // getter & setter
}
```

Bidirectional @OneToOne

- 양방향 일대일 관계
  - 객체 그래프 탐색이 자유로움
    - faculty.getSalary(), salary.getFaculty() 모두 가능
  - 관계 생명주기를 쉽게 동기화 가능
    - cascade, orphanRemoval
  - 명확한 도메인 표현력
    - Faculty가 Salary를 소유하며, Salary도 자신이 속한 Faculty를 명확히 가리킴
  - 지연 로딩은 프록시 객체로 처리되므로, 트랜잭션 바깥에서 접근 시 LazyInitializationException에 주의해야 함
  - mappedBy는 외래키를 소유하는 필드명을 정확히 명시해야 함
    - JPA에서는 항상 외래키가 있는 쪽(Salary)만이 연관관계 변경을 DB에 반영할 수 있음

Fetch

- JPA의 fetch 속성
  - 연관된 Entity를 어떻게 로딩할 것인지를 결정
  - 이는 성능 최적화와 직결되는 개념으로, @OneToOne, @OneToMany, @ManyToOne, @ManyToMany에서 사용됨

Fetch 종류	설명
FetchType.EAGER	연관된 Entity를 **즉시 로딩함** (내부적으로 Join 발생 가능) 모든 연관 객체를 항상 사용한다면 유용하지만, 그렇지 않다면 불필요한 성능 낭비
FetchType.LAZY	연관된 Entity를 **지연 로딩함** (실제로 사용할 때 쿼리 실행) 프록시 객체를 반환하며, 해당 객체를 사용하는 순간 DB에 쿼리를 날려 데이터를 가져옴 불필요한 쿼리를 피할 수 있어 성능 최적화에 유리함

```
@OneToOne(fetch = FetchType.EAGER)
private Profile profile;

@OneToMany(mappedBy = "member", fetch = FetchType.LAZY)
private List<Post> posts;
```

Fetch

- Fetch 전략 설정 가이드
  - 대부분의 관계는 LAZY로 설정하고, 명시적으로 Fetch Join으로 조회
  - 무조건 EAGER는 성능 이슈 발생

관계 유형	기본 전략	실전 권장 전략	비고
@ManyToOne	EAGER	LAZY	N+1방지
@OneToOne	EAGER	상황에 따라 결정	
@OneToMany	LAZY	유지	
@ManyToMany	LAZY	중간 엔티티로 대체	

Cascade

- 영속성 전이(Persistence Cascade)
  - 특정 Entity를 영속 상태로 만들거나 삭제를 할 때, 연관된 Entity도 함께 처리하고자 할 경우 사용
  - CascadeType.ALL, PERSIST, REMOVE 등 사용 가능
  - CascadeType.ALL 설정은 부모 엔티티 (Member)를 영속성 작업이 자식 엔티티 (Board)에 전이됨(삽입, 수정, 삭제)

```
@Entity
public class Member {
   @Id
    private Long id;
    @OneToMany(mappedBy = "member", fetch=FetchType.EAGER, cascade=CascadeType.ALL)
    private List<Board> boardList;
}
@Entity
public class Board {
   @Id
    private Long id;
    @ManyToOne
    @JoinColumn(name = “member_id")
    private Member member;
}
```

OrphanRemoval

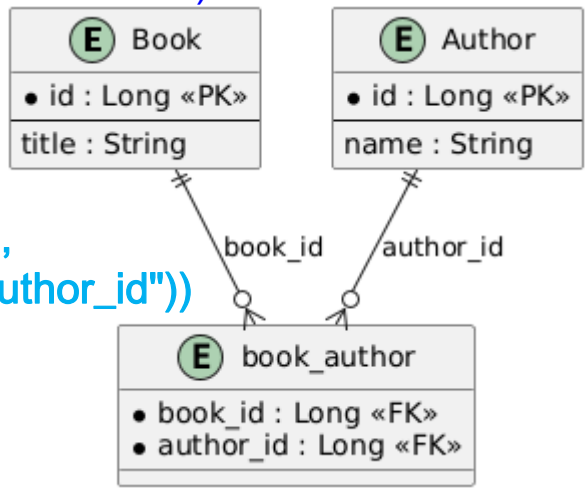
- orphanRemoval=true 속성
  - 부모 Entity (Faculty)에서 참조가 끊어졌을 때(즉, 고아가 된 경우) 자식 Entity (Salary)를 자동으로 삭제하는 데 사용됨
    - Faculty.salary = null 처럼 연관관계를 끊으면, 기존에 연결되어 있던 Salary는 자동으로 삭제됨 (DB에서 DELETE 수행됨)
    - 이는 실제 삭제 명령(entityManager.remove(salary))을 호출하지 않아도, 연결이 끊기면 JPA가 자동으로 삭제 처리한다는 뜻
  - 일대일 및 일대다 관계에서 참조 무결성(referential integrity)을 보장하고 cascade 관리를 간소화하는 데 사용되는 기능

```
faculty.setSalary(null);
// salary 객체는 고아(orphan)가 되어 DB에서 삭제됨
```

Unidirectional @ManyToMany

- 단방향 @ManyToMany 관계에서 한 Entity의 여러 instance가 다른 Entity의 여러 instance와 연결됨
  - 예로, 책과 저자 간의 관계 - 각 Book은 여러 Author를 가질 수 있고, 각 Author는 여러 Book을 작성할 수 있음

```
@Entity
public class Book {
    @Id    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    @ManyToMany
    @JoinTable(name = "book_author",
            joinColumns = @JoinColumn(name = "book_id"),
            inverseJoinColumns = @JoinColumn(name = "author_id"))
    private Set<Author> authors;
}
@Entity
public class Author {
    @Id    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
}
```



Unidirectional @ManyToMany

- N:N Book -> Author 관계
  - Book Entity만 Author를 알 수 있음 (즉, 읽기 쓰기가 필요할 때 Book 기준으로만 조작함)
  - Author는 Book이 어떤 것과 연관되었는지 모름 (즉, Author에서 Book 조회 불가)
  - JPA는 이 관계를 위해 자동으로 중간 테이블(book_author)을 생성하며, 외래키(FK) 2개를 가짐 (book_id, author_id)
    - book_author는 순수 조인 테이블 (추가 속성 없음)
    - @JoinTable 을 통해 이름과 컬럼 매핑을 명시
  - Book Entity의 authors 필드는 Set<Author>이지만, 이는 중간 테이블을 통해 간접적으로 매핑됨
  - Author 입장에서는 자신이 어떤 Book에 속하는지 알 수 없음
  - 연결 테이블에 추가 속성(예: 참여 날짜, 역할 등)을 넣고 싶을 경우 구조 확장 불가

Bidirectional @ManyToMany

- 양방향 @ManyToMany 관계는 두 Entity가 서로를 참조
  - 하나의 중간 테이블(book_author)만 생성되나, 단방향에 비해 양쪽 탐색이 가능하며, 데이터 일관성이 향상되는 장점이 있음

```
@Entity
public class Book {
    @Id    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    @ManyToMany
    @JoinTable(name = "book_author",
            joinColumns = @JoinColumn(name = "book_id"),
            inverseJoinColumns = @JoinColumn(name = "author_id"))
    private Set<Author> authors = new HashSet<>();
}
@Entity
public class Author {
    @Id    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @ManyToMany(mappedBy = "authors"))
    private Set<Book> books = new HashSet<>();
}
```

Bidirectional @ManyToMany

- N:N Book <-> Author 관계
  - 양방향 @ManyToMany는 양쪽 엔티티 모두가 서로를 참조하여, Book으로부터 Author 탐색 가능 및 Author로부터 Book 탐색 가능
  - Book이 **주 테이블(owner)** 역할을 하며 @JoinTable을 정의 (즉, JPA는 소유자(owner)인 Book 쪽의 정보만 사용하여 DB 조작을 수행함)
  - Author는 **역방향(inverse side)**이며, **mappedBy = "authors"**로 지정 (즉, mappedBy가 선언된 Author.books는 조회 용도로만 사용되며, 실제 insert/update에는 영향이 없음)

Use Intermediate Entity Instead of Direct @ManyToMany

- 실무에서는 @ManyToMany 관계는 거의 사용하지 않음
  - 추가 속성 (예: 등록일, 상태 등) 저장 불가
  - 유지보수 어려움
- 별도 매핑 엔티티로 푸는 방식이 일반적임
  - **중간 Entity 사용 (예: BookAuthor)**

```
@Entity
@Table(name = "book_author")
public class BookAuthor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @ManyToOne(optional = false, fetch = FetchType.LAZY)
    @JoinColumn(name = "book_id", nullable = false)
    private Book book;

    @ManyToOne(optional = false, fetch = FetchType.LAZY)
    @JoinColumn(name = "author_id", nullable = false)
    private Author author;
}
```

Use Intermediate Entity Instead of Direct @ManyToMany

```
@Entity
public class Book {
    @Id    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    @OneToMany(mappedBy = "book", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<BookAuthor> bookAuthors = new HashSet<>();
}
@Entity
public class Author {
    @Id    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval =
true)
    private Set<BookAuthor> bookAuthors = new HashSet<>();
}
```

N+1 Problem

- N+1 문제
  - 성능 저하의 대표적 사례 중 하나인 N+1 문제는 연관 엔티티를 조회하는 방식에서 비롯됨
  - @OneToMany, @ManyToOne 등에서 연관된 Entity를 반복적으로 조회할 때, JPA는 매번 추가 쿼리를 발생시킴
  - 예시로, Department 1개를 조회 후, 각 Department에 속한 Employee N명을 가져오는 경우, 총 1 + N 쿼리 발생

    select * from department – 1회

    select * from employee where department_id = ? – N 회

N+1 Problem

- N+1 해결 방법
  - JPQL에서 Fetch Join 사용하여 연관 엔티티를 함께 즉시 로딩 명시

```
@Query("SELECT d FROM Department d JOIN FETCH d.employees")
List<Department> findAllWithEmployees();
```

  - @EntityGraph로 정적 또는 동적으로 연관 엔티티를 로딩할 수 있도록 지정

```
@EntityGraph(attributePaths = "employees")
@Query("SELECT d FROM Department d")
List<Department> findAllWithEmployees();
```

  - @BatchSize 옵션 설정
    - 연관 엔티티들을 지정된 batch size로 묶어 조회하여 쿼리 수를 줄임
    - Lazy 로딩 방식은 유지하되, N+1 → 1+1[N/batchSize]로 개선
  - @Fetch(FetchMode.SUBSELECT) (Hibernate 전용)
    - 처음 엔티티들을 로딩한 후, 연관 컬렉션들을 서브쿼리 하나로 조회
    - 1 + 1 쿼리 구조지만, 두 번째 쿼리에서 모든 연관 데이터를 한꺼번에 조회

Infinite Recursion

- 무한 루프 문제
  - 양방향 연관관계를 사용하는 JPA 엔티티를 JSON으로 직렬화(Serialization)할 때 발생할 수 있는 대표적인 문제는 무한 루프(Infinite Recursion) 현상
  - 이는 주로 Spring Boot 환경에서 Jackson 라이브러리를 이용해 객체를 JSON으로 변환할 때 발생하며, 웹 API 응답 시 매우 빈번하게 마주칠 수 있음
  - 양방향 관계 Cart → CartItem → Cart → CartItem → 무한 반복...
  - Jackson은 객체 참조를 순회하며 JSON을 생성하는데, 이 과정에서 자기 자신을 다시 참조하게 되어 StackOverflowError 또는 서버가 응답 안함

```
@Entity
public class Cart {
    @Id @GeneratedValue
    private Long id;
    @OneToMany(mappedBy = "cart")
    private List<CartItem> items;
}
```

```
@Entity
public class CartItem {
    @Id @GeneratedValue
    private Long id;
    @ManyToOne
    @JoinColumn(name = "cart_id")
    private Cart cart;
}
```

Infinite Recursion

- DTO 분리를 통한 우회로 해결 (가장 권장)
  - 엔티티를 직접 JSON으로 반환하지 않고, 별도의 DTO 클래스를 정의하여 필요한 필드만 전달
  - 직렬화 대상에서 양방향 참조를 명시적으로 제거

```
public class CartDto {
    private Long id;
    private List<CartItemDto> items;
}
public class CartItemDto {
    private Long id;
    private String itemName;
    // Cart는 포함하지 않음!
}
public CartDto toDto(Cart cart) {
    List<CartItemDto> itemDtos = cart.getItems().stream()
        .map(item -> new CartItemDto(item.getId(), item.getName()))
        .collect(Collectors.toList());
    return new CartDto(cart.getId(), itemDtos);
}
```

Infinite Recursion

- @JsonIgnore 로 해결
  - **특정 필드를 직렬화 대상에서 강제로 제외**
  - 직렬화 방향을 제어하기 어렵고, 구조가 바뀔 때 실수할 수 있음
  - DTO 방식이 더 명확하고 유지보수 용이

```
@Entity
public class CartItem {
   @Id @GeneratedValue
    private Long id;
    @ManyToOne
    @JoinColumn(name = "cart_id")
    @JsonIgnore
    private Cart cart;
}
```

Infinite Recursion

- @JsonManagedReference (부모), @JsonBackReference (자식)로 해결
  - **한 방향만 직렬화**하고, 다른 방향은 무시하여 순환을 방지함 (보통 부모 -> 자식 방향은 직렬화하고, 자식 -> 부모 방향은 생략함)
  - 예시로 직렬화(Serialization) 과정에서 Cart는 CartItem을 직렬화하지만, 반대로 CartItem은 Cart를 직렬화하지 않음 (순환 참조 차단)

```
@Entity
public class Cart {
    @Id @GeneratedValue
    private Long id;
    @OneToMany(mappedBy = "cart")
    @JsonManagedReference
    private List<CartItem> items;
}
```

```
@Entity
public class CartItem {
    @Id @GeneratedValue
    private Long id;
    @ManyToOne
    @JoinColumn(name = "cart_id")
    @JsonBackReference
    private Cart cart;
}
```