Spring Jpa02

강사 강태광

1. JPA 연관관계

1. JPA (Java Persistence API) 연관관계 방향
 - 단방향 / 양방향.

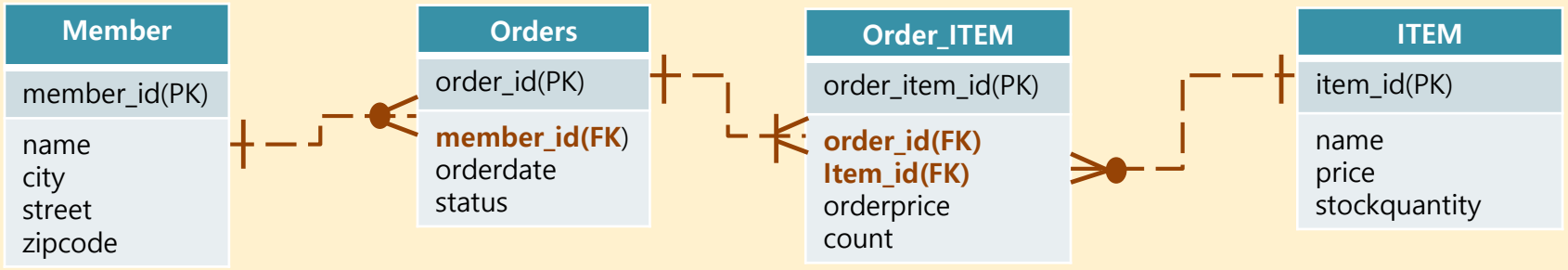
2. JPA (Java Persistence API) 연관관계 다중성
 - 다대일(N:1), 일대다(1:N), 일대일(1:1), 다대다(N:M)

3. JPA (Java Persistence API) 연관관계 Owner(주인)
 - 객체 양방향 연관관계는 관리 Owner(주인) 필요

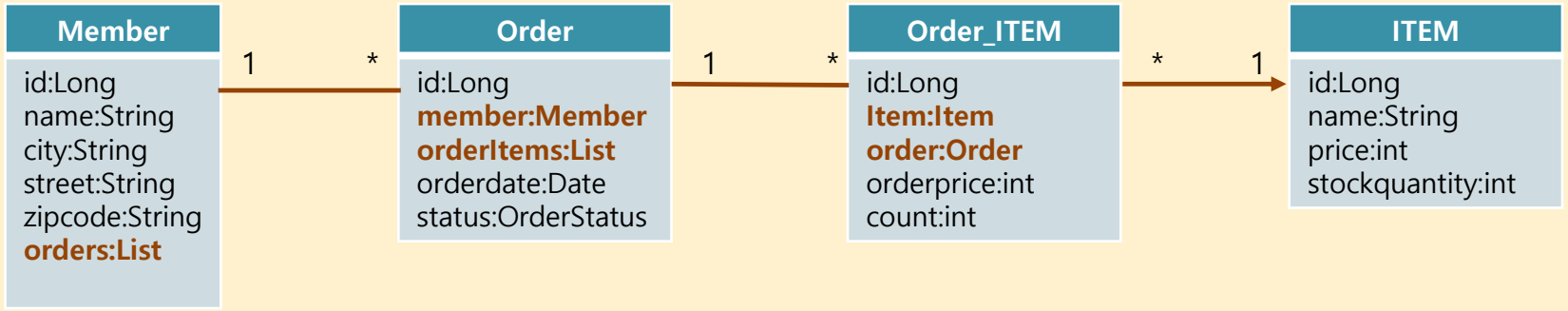
2. JPA 연관관계 시나리오

1. JPA (Java Persistence API) 연관관계 시나리오
   - 회원과 팀이 있다.
   - 회원은 하나의 팀에만 소속.
   - 회원과 팀은 다대일 관계

Table 연관관계



객체 연관관계



3-1. JPA 연관관계 다중성

1. JPA (Java Persistence API) 다중성
   - 다대일(@ManyToOne)
   - 일대다(@OneToMany)
   - 일대일@OneToOne)
   - 다대다(@ManyToMany)

2.다중성에 대한 단방향/양방향

Table	객체
외래Key 하나로 양쪽 Join 가능	참조용 필드가 있는 쪽으로만 참조 가능 한쪽만 참조하면 단방향 양쪽이 서로 참조하면 양방향

3. 연관관계 Owner
   1) 테이블은 외래 키 하나로 두 테이블이 연관관계를 맺음
   2) 연관관계의 주인: 외래 키를 관리하는 참조
   3) 주인의 반대편: 외래 키에 영향을 주지 않음, 단순 조회만 가능

4. 객체 연관관계 - 다대일

- 외래 키가 있는 쪽이 연관관계의 주인
- 양쪽을 서로 참조하도록 개발
- Project 권장
- 가장 많이 사용하는 연관관계(다대일 단방향)

3-2. JPA 연관관계 다중성

5. 객체 연관관계 - 일대다 Table 연관관계

- 일대다 단방향은 일대다(1:N)에서 일(1)이 연관관계의 주인
- 테이블 일대다 관계는 항상 다(N) 쪽에 외래 키가 있음
- 객체와 테이블의 차이 때문에 반대편 테이블의 외래 키를 관리하 는 특이한 구조
- @JoinColumn을 꼭 사용해야 함. 그렇지 않으면 조인 테이블 방식을 사용함(중간에 테이블을 하나 추가함)
- 문제점
  ① 엔티티가 관리하는 외래 키가 다른 테이블에 있음
  ② 연관관계 관리를 위해 추가로 UPDATE SQL 실행
  ③ 일대다 단방향 매핑보다는 다대일 양방향 매핑을 사용 권장
  ④ 일대다 양방향 , 이런 매핑은 공식적으로 존재하지 않음
  ⑤ 읽기 전용 필드를 사용해서 양방향 처럼 사용하는 방법, @JoinColumn(insertable=false, updatable=false)

3-3. JPA 연관관계 다중성

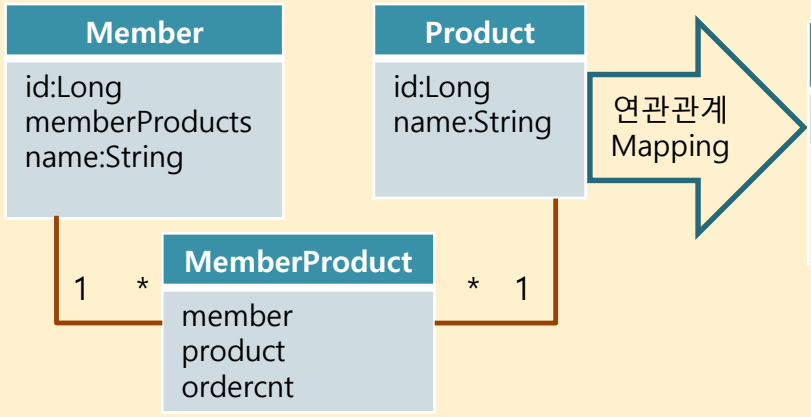
6. 객체 연관관계 - 일대일 Table 연관관계

1) 주 테이블에 외래 키
- 주 객체가 대상 객체의 참조를 가지는 것 처럼
- 주 테이블에 외래 키를 두고 대상 테이블을 찾음
- 객체지향 개발자 선호
- JPA 매핑 편리
  장점: 주 테이블만 조회해도 대상 테이블에 데이터가 있는지 확인 가능
  단점: 값이 없으면 외래 키에 null 허용

2) 대상 테이블에 외래 키
- 대상 테이블에 외래 키가 존재
- 전통적인 데이터베이스 개발자 선호
- 장점: 주 테이블과 대상 테이블을 일대일에서 일대다 관계로 변경할 때 테이블 구조 유지
- 단점: 프록시 기능의 한계로 지연 로딩으로 설정해도 항상 즉시 로딩

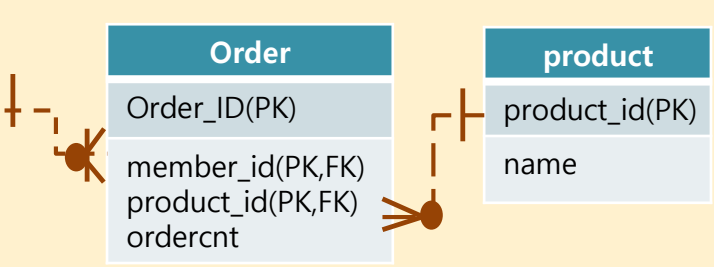
3-4. JPA 연관관계 다중성

7. 객체 연관관계 - 다대다



연관관계 Mapping

Table 연관관계



1) 관계형 데이터베이스는 정규화된 테이블 2개로 다대다 관계를 표현할 수 없음
2) 연결 테이블을 추가해서 일대다, 다대일 관계로 풀어내야함
3) 실무에서 거의 사용하지 않음
4) 다대다 사용하기 위해 연결 테이블용 엔티티 추가(연결 테이블을 엔티티로 승격)
   - @ManyToMany -> @OneToMany, @ManyToOne

4-1. JPA 연관관계 어노테이션 주요 속성

1. 외래 키를 매핑할 때 사용하는 @Joincolumn

속성	내 용	기본값
name	매핑할 외래 키 이름	필드명 + _ + 참조하는 테이블의 기본 키 컬럼명
referencedColumnName	외래 키가 참조하는 대상 테이블의 컬럼명	참조하는 테이블의 기본 키 컬럼명
foreignKey(DDL)	외래 키 제약조건을 직접 지정할 수 있음 이 속성은 테이블을 생성할 때만 사용.	운영DB에는 사용 안됨, validate 또는 none
unique nullable insertable updatable columnDefinition table	@Column의 속성과 같음	

2. 다대일 관계 매핑 @ManyToOne

속성	내 용	기본값
optional	false로 설정하면 연관된 엔티티가 항상 있어야 한다	TRUE
fetch	글로벌 페치 전략을 설정	@ManyToOne=FetchType.EAGER @OneToMany=FetchType.LAZY
foreignKey(DDL)	외래 키 제약조건을 직접 지정할 수 있음 이 속성은 테이블을 생성할 때만 사용.	운영DB에는 사용 안됨, validate 또는 none
cascade	영속성 전이 기능을 사용	
targetEntity	연관된 엔티티의 타입 정보를 설정. 컬렉션을 사용해도 제네릭으로 타 입 정보를 알 수 있음	이 기능은 거 의 사용하지 않음

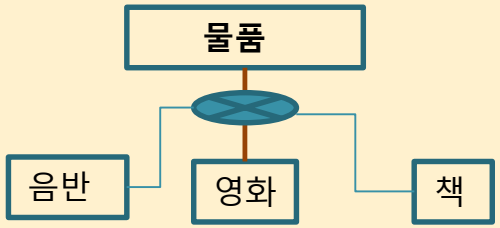
4-2. JPA 연관관계 어노테이션 주요 속성

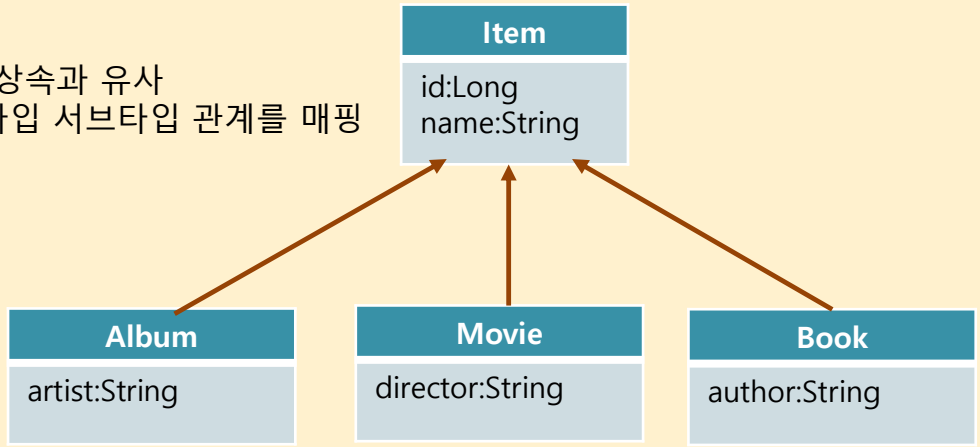
3. 일대다 관계 매핑 @OneToMany

속성	내 용	기본값
mappedBy	연관관계의 주인 필드를 선택	
fetch	글로벌 페치 전략을 설정	@ManyToOne=FetchType.EAGER @OneToMany=FetchType.LAZY
foreignKey(DDL)	외래 키 제약조건을 직접 지정할 수 있음 이 속성은 테이블을 생성할 때만 사용.	운영DB에는 사용 안됨, validate 또는 none
cascade	영속성 전이 기능을 사용	
targetEntity	연관된 엔티티의 타입 정보를 설정. 컬렉션을 사용해도 제네릭으로 타 입 정보를 알 수 있음	이 기능은 거 의 사용하지 않음

5. JPA 연관관계 상속관계 mapping

1. RDB 상속관계
   1) 관계형 데이터베이스는 상속 관계 없음
   2) 슈퍼타입 서브타입 관계라는 모델링 기법이 객체 상속과 유사
   3) 상속관계 매핑: 객체의 상속과 구조와 DB의 슈퍼타입 서브타입 관계를 매핑





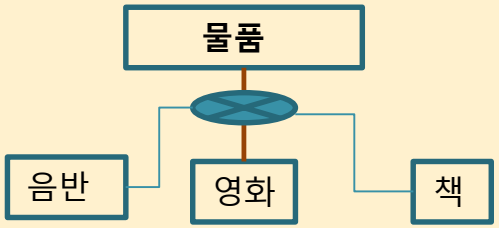
2. 슈퍼타입 서브타입 논리 모델을 실제 물리 모델로 구현 전략
   1) 각각 Table로 변환 -> Join 전략
   2) 통합 Table로 변환 -> Single Table 전략
   3) 서브타입 Table로 변환 -> 구현 클래스마다 Table 전략

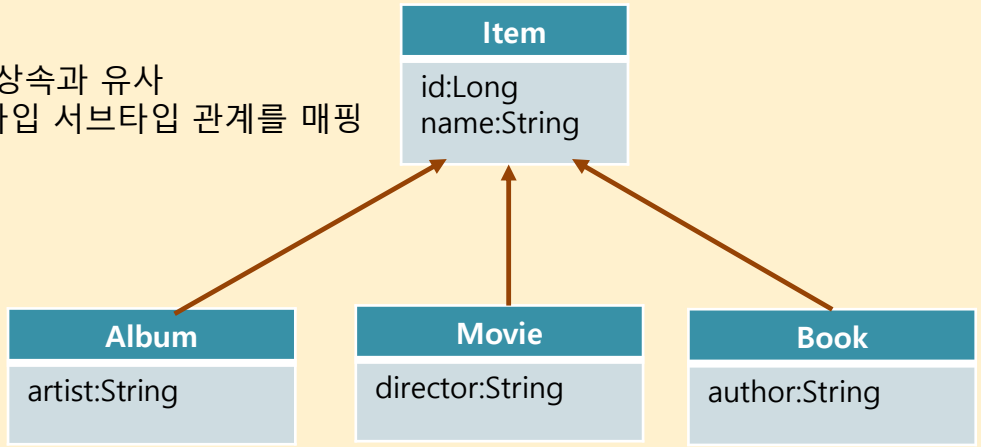
3. 주요 annotation

속성	내 용
@Inheritance(strategy=InheritanceType.XXX)	• JOINED: 조인 전략 • SINGLE_TABLE: 단일 테이블 전략 • TABLE_PER_CLASS: 구현 클래스마다 테이블 전략
@DiscriminatorColumn(name="DTYPE")	**어떤 컬럼을 가지고 어떤 자식 엔터티를 판별할 것인가**에 대한 힌트를 주는 어노테이션
@DiscriminatorValue("XXX")	@DiscriminatorColumn(name = "dtype"), 즉 dtype 필드의 값이 뭐냐에 따라서 어떤 엔터티를 사용할지가 정해짐

5-1. JPA 연관관계 상속관계 mapping

1. RDB 상속관계
   1) 관계형 데이터베이스는 상속 관계 없음
   2) 슈퍼타입 서브타입 관계라는 모델링 기법이 객체 상속과 유사
   3) 상속관계 매핑: 객체의 상속과 구조와 DB의 슈퍼타입 서브타입 관계를 매핑





2. 슈퍼타입 서브타입 논리 모델을 실제 물리 모델로 구현 전략
   1) 각각 Table로 변환 -> Join 전략
   2) 통합 Table로 변환 -> Single Table 전략
   3) 서브타입 Table로 변환 -> 구현 클래스마다 Table 전략

3. 주요 annotation

속성	내 용
@Inheritance(strategy=InheritanceType.XXX)	• JOINED: 조인 전략 • SINGLE_TABLE: 단일 테이블 전략 • TABLE_PER_CLASS: 구현 클래스마다 테이블 전략
@DiscriminatorColumn(name="DTYPE")	**어떤 컬럼을 가지고 어떤 자식 엔터티를 판별할 것인가**에 대한 힌트를 주는 어노테이션
@DiscriminatorValue("XXX")	@DiscriminatorColumn(name = "dtype"), 즉 dtype 필드의 값이 뭐냐에 따라서 어떤 엔터티를 사용할지가 정해짐

5-2. JPA 연관관계 상속관계 mapping

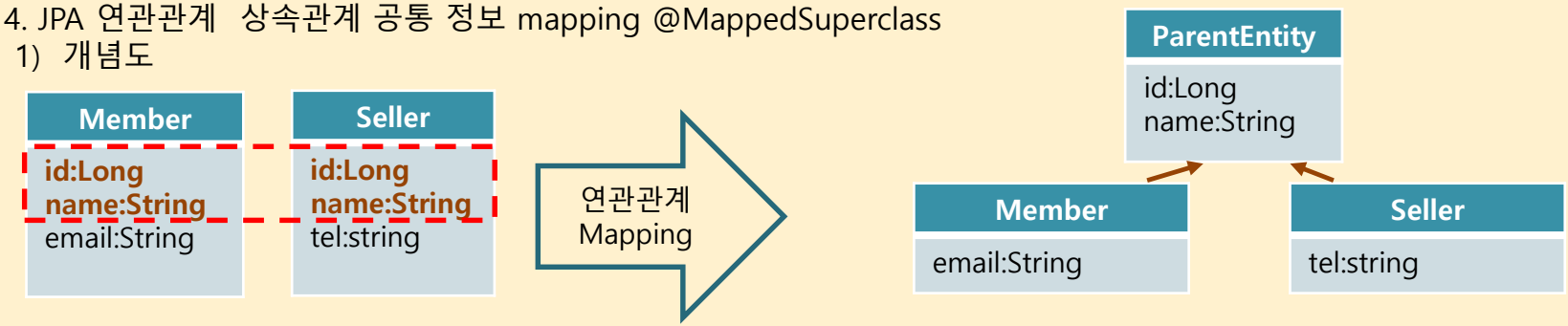
4. JPA 연관관계 상속관계 mapping 전략의 장단점

전략	장점	단점
조인 (각각 Table)	• Table 정규화 • 외래 키 참조 무결성 제약조건 활용가능 • 저장공간 효율화	• 조회시 조인을 많이 사용, 성능 저하 • 조회 쿼리가 복잡함 • 데이터 저장시 INSERT SQL 2번 호출
단일 Table	• **조인이 필요 없으므로 일반적으로 조회 성능이 빠름** • **조회 쿼리가 단순함**	• 자식 Entity가 매핑한 Column은 모두 null 허용 • 단일 Table에 모든 것을 저장하므로 Table이 커질 수 있음. • 상황에 따라서 조회 성능이 오히려 느려질 수 있음
구현 Class마다 Table	• **서브 타입**을 명확하게 구분해서 처리할 때 효과적 • not null 제약조건 사용 가능	• 이 전략은 데이터베이스 설계자와 ORM 전문가 둘 다 추천하지 않음 • 여러 자식 테이블을 통합(함께)조회할 때 성능이 느림(UNION SQL 필요)

5-3. JPA 연관관계 공통 정보 mapping

4. JPA 연관관계 상속관계 공통 정보 mapping @MappedSuperclass

1) 개념도



2) 주요 내용

- 부모 클래스를 상속 받는 자식 클래스에 매핑 정보만 제공
- 조회, 검색 불가(em.find(ParentEntity) 불가)
- 직접 생성해서 사용할 일이 없으므로 추상 클래스 권장
- 테이블과 관계 없고, 단순히 엔티티가 공통으로 사용하는 매핑정보를 모으는 역할
- 주로 등록일, 수정일, 등록자, 수정자 같은 전체 엔티티에서 공통으로 적용하는 정보를 모을 때 사용
- @Entity 클래스는 엔티티나 @MappedSuperclass로 지정한 클래스만 상속 가능

6-1. JPA 연관관계 관리 , Proxy

1. Proxy 개념

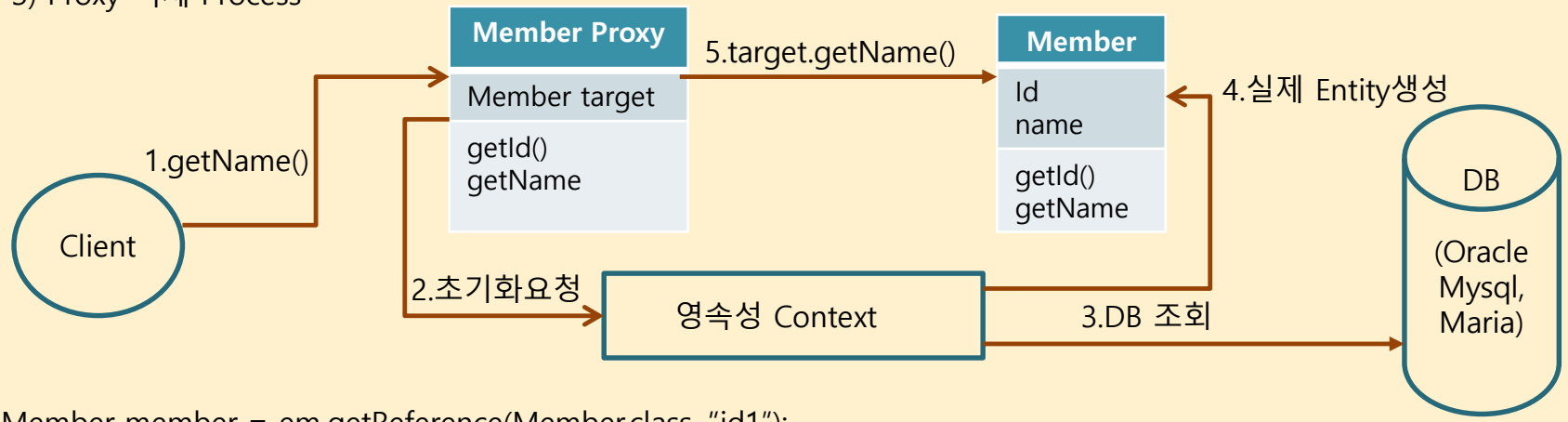
1) 주요 개념

- 실제 클래스를 상속 받아서 만들어지며 실제 클래스와 겉 모양이 같다.
- 사용하는 입장에서는 진짜 객체인지 프록시 객체인지 구분하지 않고 사용하면 됨
- Proxy 객체는 실제 객체의 참조(target)를 보관
- Proxy 객체를 호출하면 Proxy 객체는 실제 객체의 메소드 호출

2) 주요 내용

- 부모 클래스를 상속 받는 자식 클래스에 매핑 정보만 제공
- 조회, 검색 불가(em.find(ParentEntity) 불가)
- 직접 생성해서 사용할 일이 없으므로 추상 클래스 권장
- 테이블과 관계 없고, 단순히 엔티티가 공통으로 사용하는 매핑정보를 모으는 역할
- 주로 등록일, 수정일, 등록자, 수정자 같은 전체 엔티티에서 공통으로 적용하는 정보를 모을 때 사용
- @Entity 클래스는 엔티티나 @MappedSuperclass로 지정한 클래스만 상속 가능
- 프록시 객체는 실제 객체의 참조(target)를 보관
- 프록시 객체를 호출하면 프록시 객체는 실제 객체의 메소드 호출

3) Proxy 객체 Process




```
Member member = em.getReference(Member.class, "id1");
member.getName();
```

6-2. JPA 연관관계 관리 , Proxy

2. Proxy 특징 및 관련 method

1) 주요 특징

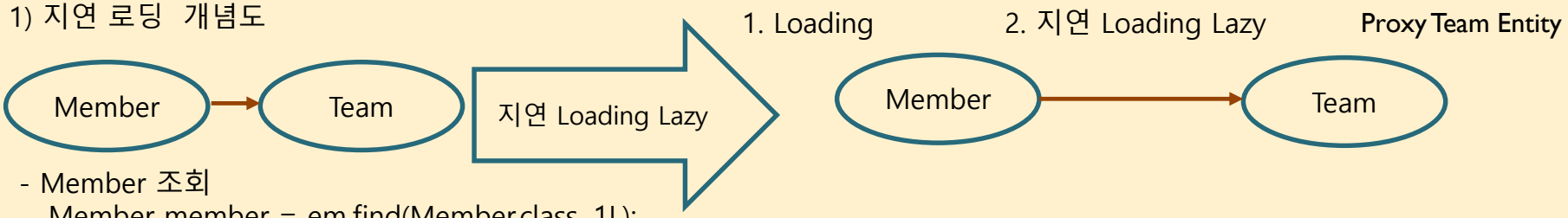
- 프록시 객체는 처음 사용할 때 한 번만 초기화
- 프록시 객체를 초기화 할 때, 프록시 객체가 실제 엔티티로 바뀌는 것은 아님, 초기화되면 프록시 객체를 통해서 실제 엔티티에 접근 가능
- 프록시 객체는 원본 엔티티를 상속받음, 따라서 타입 체크시 주의해야함 (== 비교 실패, 대신 instance of 사용)
- 영속성 컨텍스트에 찾는 엔티티가 이미 있으면 em.getReference()를 호출해도 실제 엔티티 반환
- 영속성 컨텍스트의 도움을 받을 수 없는 준영속 상태일 때, 프록시를 초기화하면 문제 발생 (하이버네이트는 org.hibernate.LazyInitializationException 예외를 터트림)

2) Proxy 확인 method

method	내 용
PersistenceUnitUtil.isLoaded(Object entity)	• 프록시 인스턴스의 초기화 여부 확인
entity.getClass().getName() 출력	• 프록시 클래스 확인 방법
org.hibernate.Hibernate.initialize(entity);	• 프록시 강제 초기화 • JPA 표준은 강제 초기화 없음 → 강제 호출: member.getName()

6-3. JPA 연관관계 관리 , 즉시 로딩과 지연 로딩

1. 지연 로딩 LAZY을 사용해서 Proxy로 조회
 1) 지연 로딩 개념도



 - Member 조회
  Member member = em.find(Member.class, 1L);
 - Team Get (실제 가져옴)
  Team team = member.getTeam();
  team.getName(); // 실제 team을 사용하는 시점에 초기화(DB 조회)
 - **실무에서는 가급적 지연 로딩만 사용**
 - **@ManyToOne, @OneToOne은 기본이 즉시 로딩 -> LAZY로 설정**
- @OneToMany, @ManyToMany는 기본이 지연 로딩
 2) 즉시 로딩
 - Member와 Team을 자주 함께 사용 할 때 (Member조회시 항상 Team도 조회)
 - 즉시 로딩 EAGER를 사용해서 함께 조회
 - JPA 구현체는 가능하면 조인을 사용해서 SQL 한번에 함께 조회
 - 즉시 로딩을 적용하면 예상하지 못한 SQL이 발생 가능성 존재

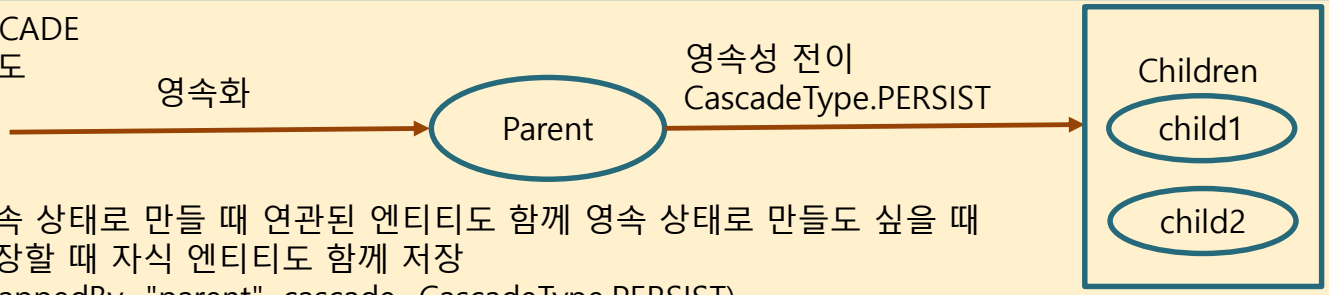
```
@Entity
 public class Member {
    @Id
    @GeneratedValue
    private Long id;
    @Column(name = "USERNAME")
    private String name;
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "TEAM_ID")
    private Team team;
    ..
}
```

1. 즉시 로딩과 지연 로딩 활용
 - Member와 Team은 자주 함께 사용 -> 즉시 로딩
 - Member와 Order는 가끔 사용 -> 지연 로딩

6-4. JPA 영속 상태 관리 , 영속성 전이 CASCADE

1. 영속성 전이 CASCADE

1) 영속성 전이 개념도



- 특정 엔티티를 영속 상태로 만들 때 연관된 엔티티도 함께 영속 상태로 만들도 싶을 때
- 부모 엔티티를 저장할 때 자식 엔티티도 함께 저장
  @OneToMany(mappedBy="parent", cascade=CascadeType.PERSIST)
- 영속성 전이는 연관관계를 매핑하는 것과 아무 관련이 없음
- 엔티티를 영속화할 때 연관된 엔티티도 함께 영속화하는 편리함 을 제공

2) CASCADE의 종류

- ALL: 모두 적용
- PERSIST: 영속
- REMOVE: 삭제
- MERGE: 병합
- REFRESH: REFRESH
- DETACH: DETACH

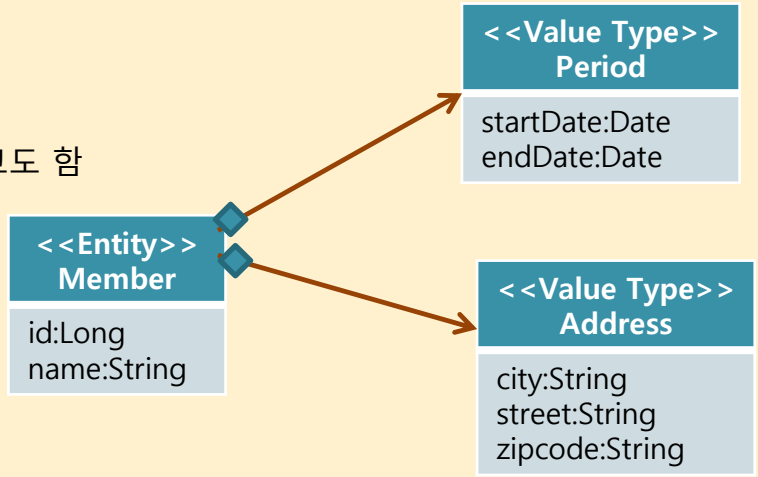
3) 고아 객체

- 참조가 제거된 엔티티는 다른 곳에서 참조하지 않는 고아 객체로 보고 삭제하는 기능
- 참조하는 곳이 하나일 때 사용해야함
- 특정 엔티티가 개인 소유할 때 사용
- @OneToOne, @OneToMany만 가능
- 참고: 개념적으로 부모를 제거하면 자식은 고아가 됨(CascadeType.REMOVE처럼 동작)
  따라서 고아 객체 제거 기능을 활성화 하면, 부모를 제거할 때 자식도 함께 제거

7-1. JPA 값 Type 유형

1. 기본 값 Type
   1) 기본 값 Type 유형
      - 자바 기본 타입(int, double)
      - 래퍼 클래스(Integer, Long)
      - String
   2) 기본 값 Type 적용
      - 생명주기를 엔티티의 의존 -> 예) 회원을 삭제하면 이름, 나이 필드도 함께 삭제
      - 값 타입은 공유하면 안 됨 -> 예) 회원 이름 변경시 다른 회원의 이름도 함께 변경되면 안됨
      - Integer같은 래퍼 클래스나 String 같은 특수한 클래스는 공유 가능한 객체이지만 변경 안됨

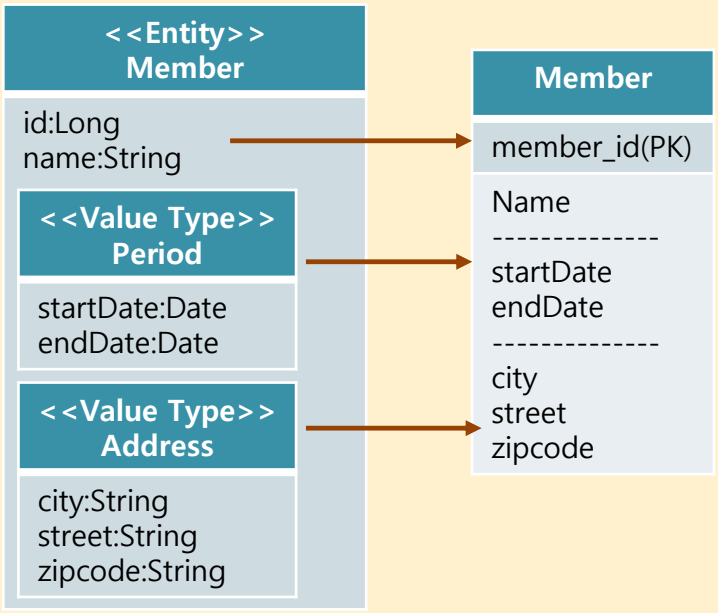
2. 임베디드 Type (복합 값 타입)
   1) 임베디드 Type 개념
      - 새로운 값 타입을 직접 정의할 수 있음
      - JPA는 임베디드 타입(embedded type)이라 함
      - 주로 기본 값 타입을 모아서 만들어서 복합 값 타입이라고도 함
      - int, String과 같은 값 타입
   2) 임베디드 Type 적용 개념도



   3) 임베디드 Type 사용법
      - @Embeddable: 값 타입을 정의하는 곳에 표시
      - @Embedded: 값 타입을 사용하는 곳에 표시
      - 기본 생성자 필수
   4) 임베디드 타입의 장점
      - 재사용
      - 높은 응집도
      - Period.isWork()처럼 해당 값 타입만 사용하는 의미 있는 메소드를 만들 수 있음
      - 임베디드 타입을 포함한 모든 값 타입은, 값 타입을 소유한 엔티티에 생명주기를 의존

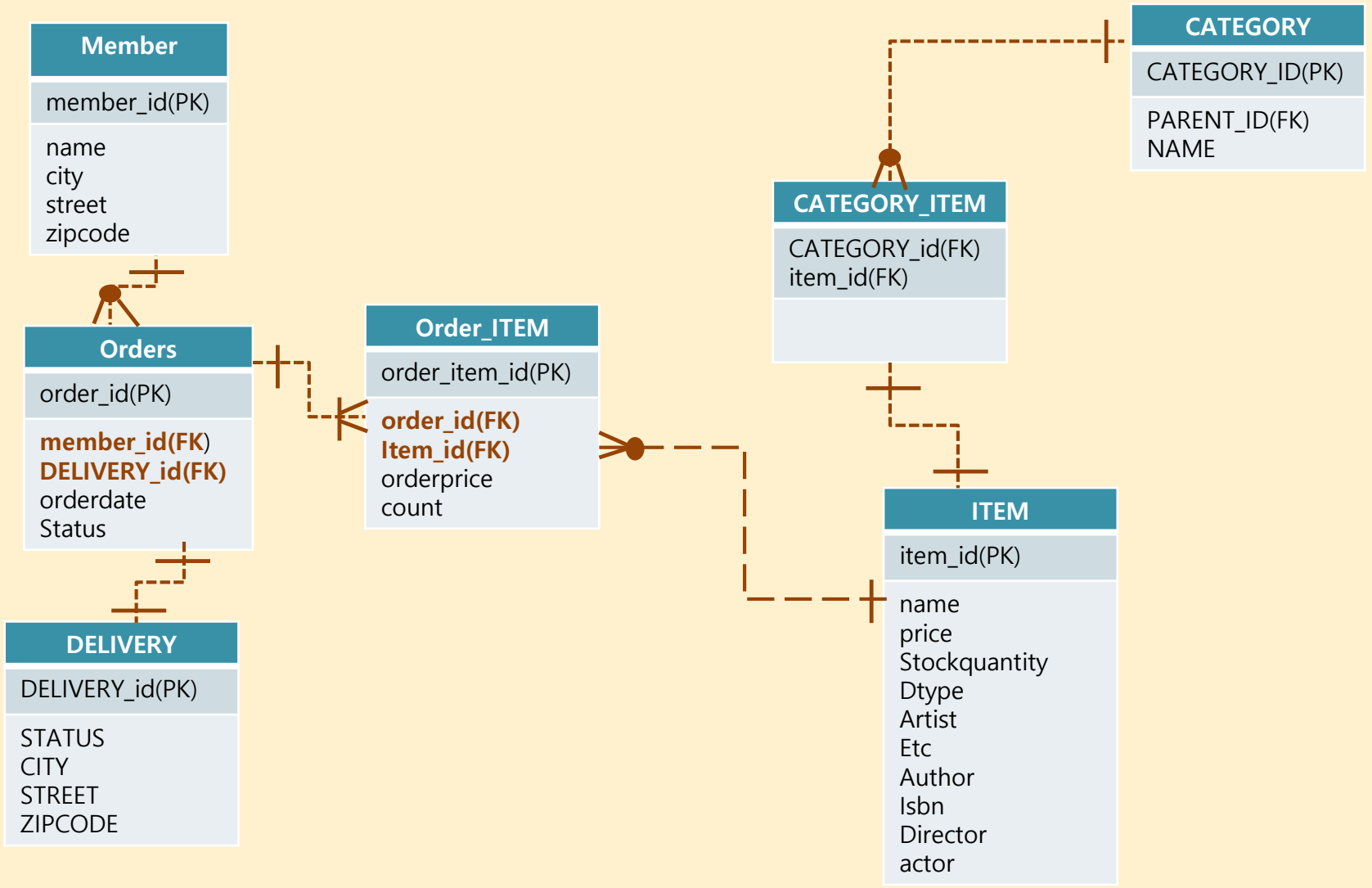
7-2. JPA 값 Type 유형

5) 임베디드 타입과 테이블 매핑
- 임베디드 타입은 엔티티의 값.
- 임베디드 타입을 사용하기 전과 후에 매핑하는 테이블은 같음
- 객체와 테이블을 아주 세밀하게(find-grained) 매핑하는 것이 가능
- 잘 설계한 ORM 애플리케이션은 매핑한 테이블의 수보다 클래스의 수가 더 많음
- 임베디드 타입 같은 값 타입을 여러 엔티티에서 공유하면 위험
  → 대신 값(인스턴스)를 복사해서 사용
- 임베디드 타입처럼 직접 정의한 값 타입은 자바의 기본 타입이 아니라 객체 타입이므로 부작용(side effect) 발생



8-1. JPA modeling

1. Table modeling(Table 연관관계)



8-2. JPA modeling

1. 객체 연관관계 modeling