

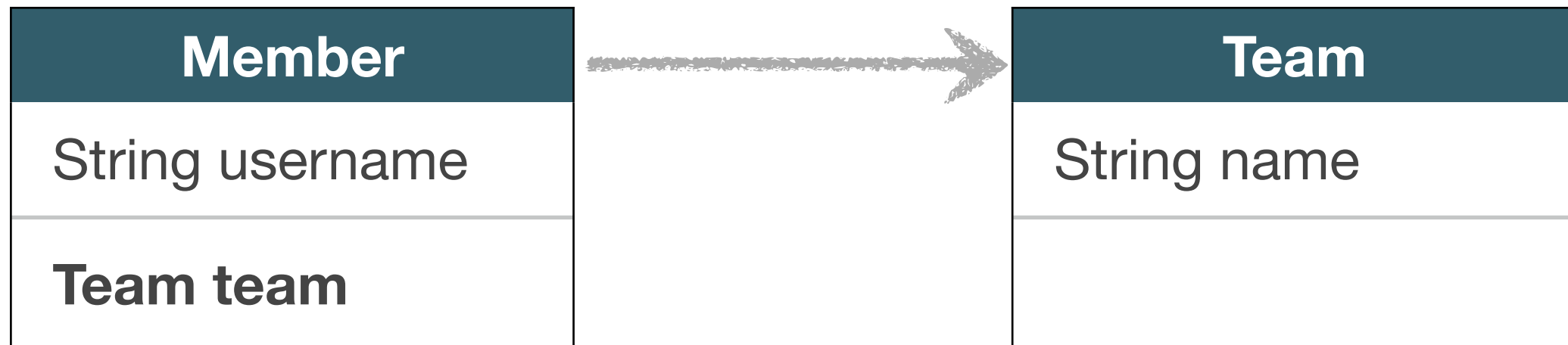
프록시와 연관관계 관리

목차

- 프록시
- 즉시 로딩과 지연 로딩
- 지연 로딩 활용
- 영속성 전이: CASCADE
- 고아 객체
- 영속성 전이 + 고아 객체, 생명주기
- 실전 예제 - 5.연관관계 관리

프록시

Member를 조회할 때 Team도 함께 조회해야 할까?



Member를 조회할 때 Team도 함께 조회해야 할까?

회원과 팀 함께 출력

이렇게 자주 쓴다면 회원을 가지고 올 때
팀도 같이 가지고 오면 좋을 것이고

```
public void printUserAndTeam(String memberId) {  
    Member member = em.find(Member.class, memberId);  
    Team team = member.getTeam();  
    System.out.println("회원 이름: " + member.getUsername());  
    System.out.println("소속팀: " + team.getName());  
}
```

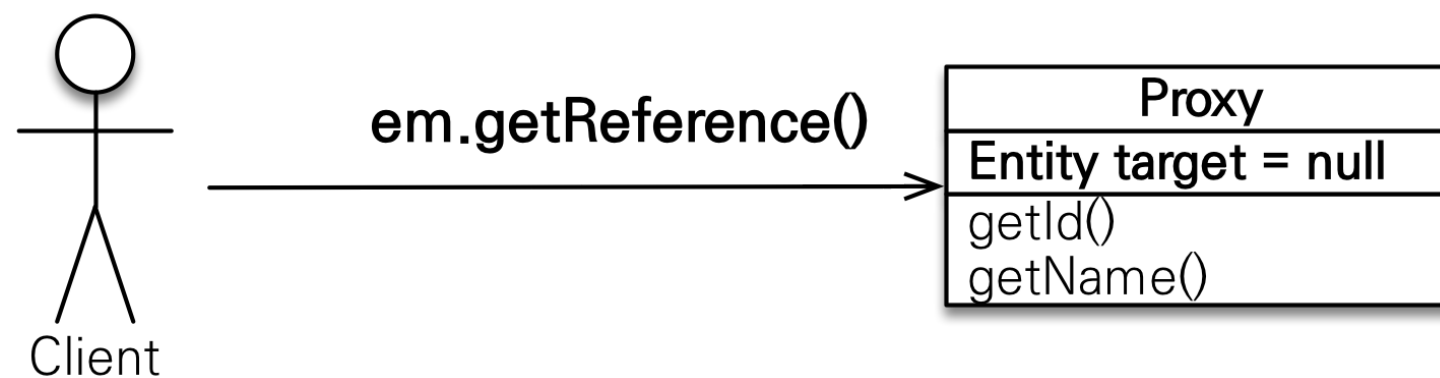
회원만 출력

이렇게 자주 쓴다면 굳이 팀 테이블까지 조회 해 올 필요는 없다. 쿼리 낭비.

```
public void printUser(String memberId) {  
    Member member = em.find(Member.class, memberId);  
    Team team = member.getTeam();  
    System.out.println("회원 이름: " + member.getUsername());  
}
```

프록시 기초

- `em.find()` vs `em.getReference()`
- `em.find()`: 데이터베이스를 통해서 실제 엔티티 객체 조회
- `em.getReference()`: 데이터베이스 조회를 미루는 가짜(프록시) 엔티티 객체 조회



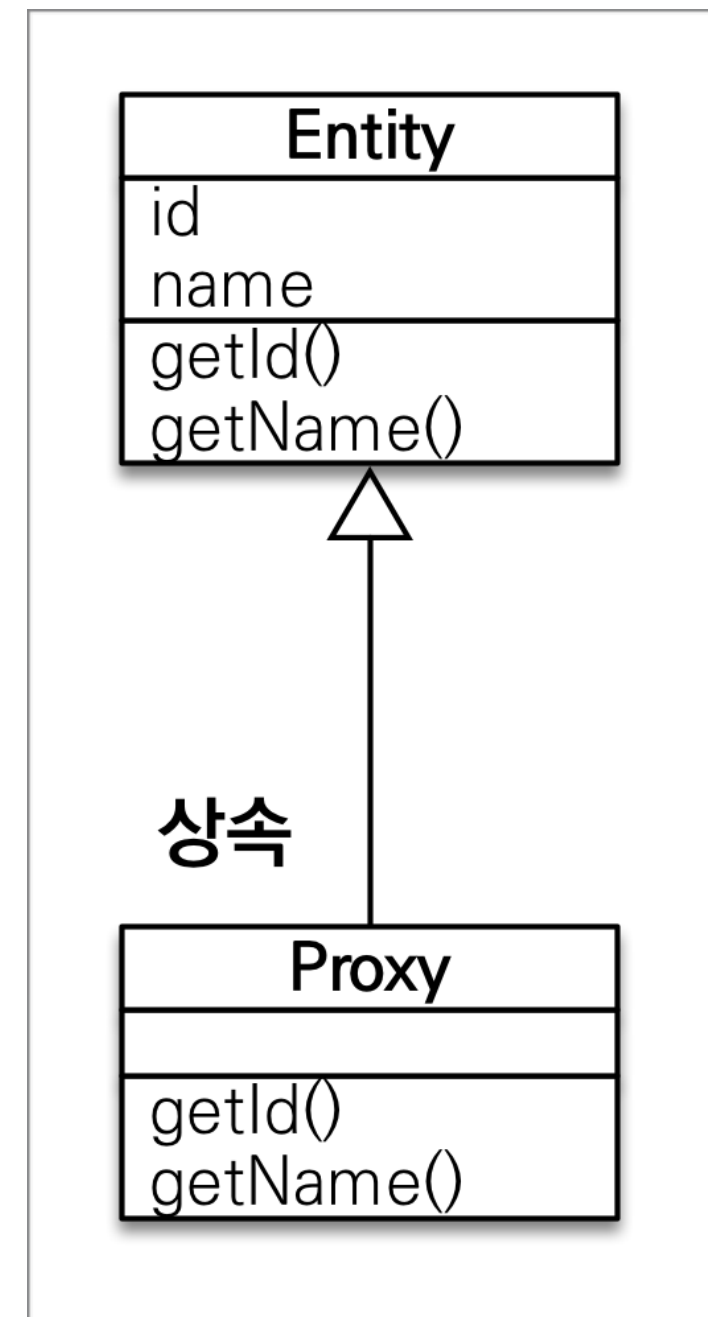
겉데기는 똑같은데 내용물이 텅텅 빈 객체
Target 은 진짜 레퍼런스를 가리킨다
(처음에는 null)

프록시 특징

Member 인스턴스나, MemberProxy 인스턴스나

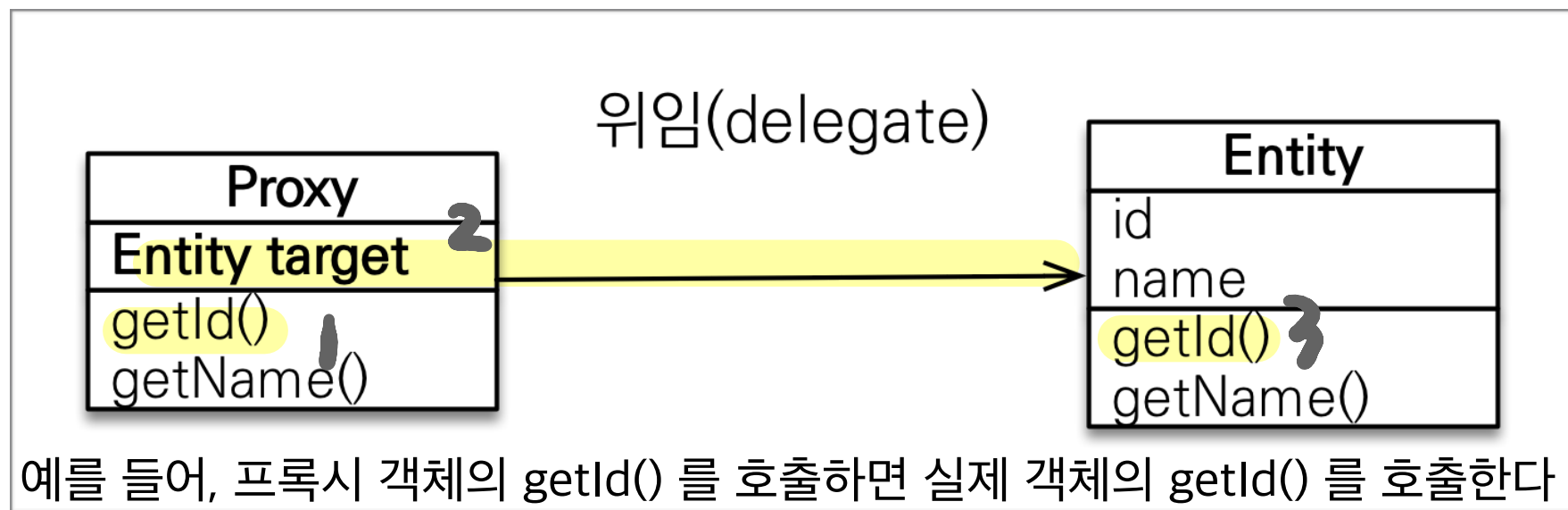
Member 데이터 타입으로 받을 수 있어서 생기는 문제 주의!

- 실제 클래스를 상속 받아서 만들어짐
- 실제 클래스와 겉 모양이 같다.
- 사용하는 입장에서는 진짜 객체인지
프록시 객체인지 구분하지 않고
사용하면 됨(이론상)



프록시 특징

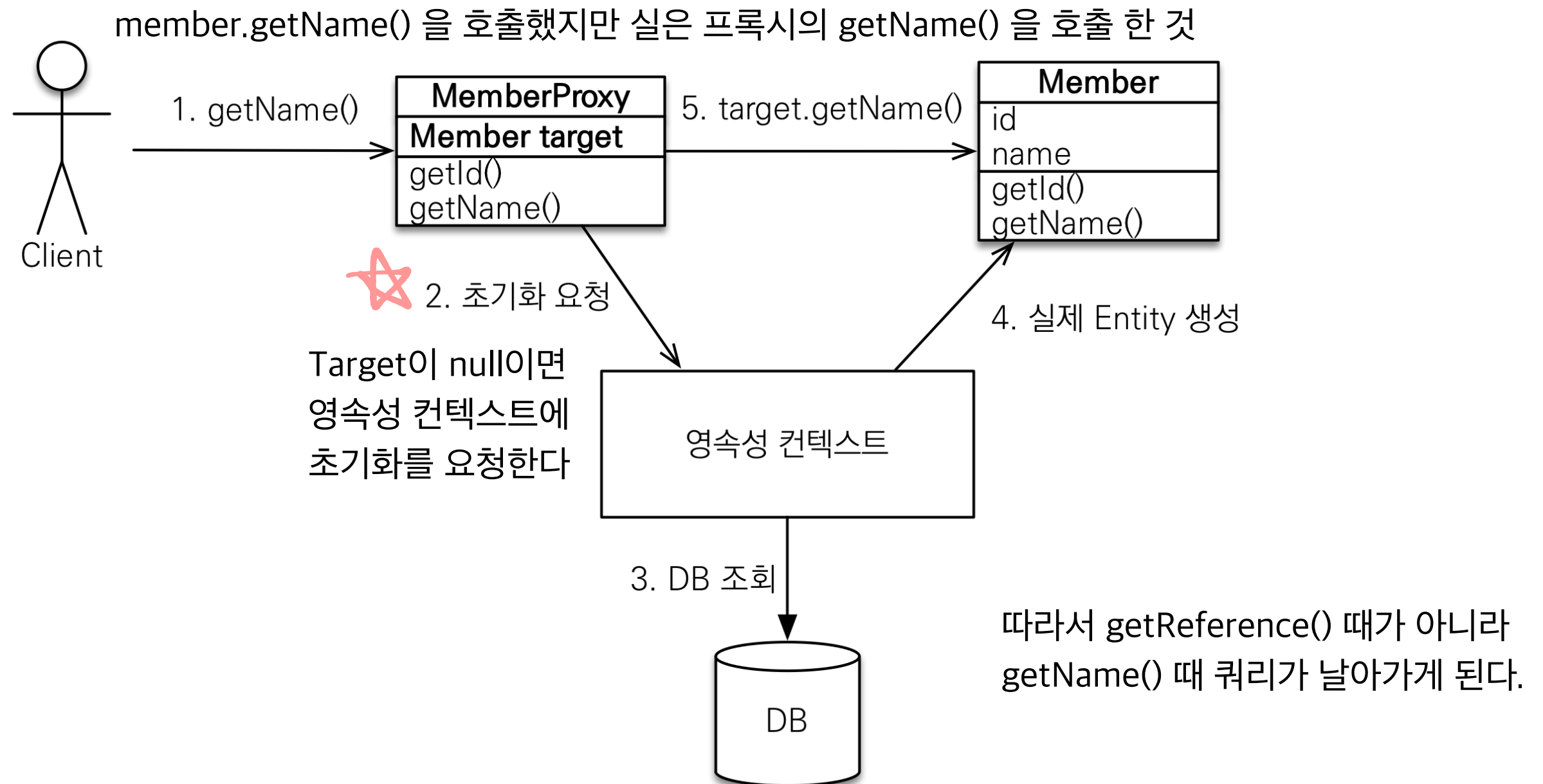
- 프록시 객체는 실제 객체의 참조(target)를 보관
- 프록시 객체를 호출하면 프록시 객체는 실제 객체의 메소드 호출



프록시 객체의 초기화

getReference() 당시는 초기화 되지 않고 getName() 시 초기화를 요청한다

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



proxy.getUsername() 을 두번 호출하면 쿼리가 두번 날아갈까?


```
1 Member findMember = em.find(Member.class, member.getId());  
2 System.out.println("findMember.username = " + findMember.getUsername());  
System.out.println("findMember.username = " + findMember.getUsername());
```

```
team1_.lastModifiedBy as lastModi4_7_1_,  
team1_.lastModifiedDate as lastModi5_7_1_,  
team1_.name as name6_7_1_  
from  
Member member0_  
left outer join  
Team team1_  
on member0_.TEAM_ID=team1_.TEAM_ID  
where  
member0_.MEMBER_ID=?  
findMember.username = hello - 1  
findMember.username = hello - 2
```

프록시 객체의 target 이 null 이 아니므로

프록시 객체의 getUsername() 을 2번째 호출시에는 쿼리가 날아가지 않는다

프록시의 특징

- 프록시 객체는 처음 사용할 때 한 번만 초기화
 - 초기화 전과 동일한 프록시 객체의 target 값이 채워지는 거지 프록시가 바뀌지는 않는다 
- 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 반대로 마찬가지로. 이미 프록시를 가져온 상태면 `em.find()` 해도 프록시가 나온다
- 영속성 컨텍스트의 도움을 받을 수 없는 준영속 상태일 때, 프록시를 초기화하면 문제 발생
(하이버네이트는 `org.hibernate.LazyInitializationException` 예외를 터트림)

타입 비교를 절대 == 로 하면 안되고,
instance of 를 사용해야 한다

```
(Member m1, Member m2) {  
1 == m2: " + (m1 instanceof Member));  
1 == m2: " + (m2 instanceof Member));  
}
```

```
Member m1 = em.find(Member.class, member1.getId());  
Member m2 = em.find(Member.class, member2.getId());  
  
System.out.println("m1 == m2: " + (m1.getClass() == m2.getClass()));
```

True

```
Member m1 = em.find(Member.class, member1.getId());  
Member m2 = em.getReference(Member.class, member2.getId());  
  
System.out.println("m1 == m2: " + (m1.getClass() == m2.getClass()));
```

False

근데 실제로는 아래같이 쓰겠지, 이 때 주의!

Member 라고 데이터 타입이 적혀 있어도
Member 일 수도 있고 프록시일 수도 있는 것!

```
private static void logic(Member m1, Member m2) {  
    System.out.println("m1 == m2: " + (m1.getClass() == m2.getClass()));  
}
```

프록시가 Member 를 상속받고 있으므로,
한쪽엔 엔티티, 한쪽엔 프록시를 넘기면 false 가 나온다

이미 영속성 컨텍스트에 존재하면 getReference() 시 프록시가 아니라 실제 엔티티를 반환한다

```
Member m1 = em.find(Member.class, member1.getId());
System.out.println("m1 = " + m1.getClass());

Member reference = em.getReference(Member.class, member1.getId());
System.out.println("reference = " + reference.getClass());
```

```
m1 = class hellojpa.Member
reference = class hellojpa.Member
```

em.getReference() 로 가져왔는데 프록시가 아니라 엔티티 클래스로 나옴!

1. 이미 멤버를 영속성 컨텍스트에 올려 놔는데, 1차 캐시에 올려놔는데 이를 프록시로 가지고 와봤자 아무 성능상 이점이 없다.
2. JPA에서는 무조건 한 영속성 컨텍스트에서 가지고 온 객체의 PK 가 똑같으면 항상 두 객체는 같아야 한다

```
System.out.println("a == a: " + (m1 == reference));
```

 항상 true여야 하는 것

```
Member refMember = em.getReference(Member.class, member1.getId());  
System.out.println("refMember = " + refMember.getClass()); //Proxy  
  
Member findMember = em.find(Member.class, member1.getId());  
System.out.println("findMember = " + findMember.getClass()); //Member  
  
System.out.println("refMember == findMember: " + (refMember == findMember));
```

JPA에서는 한 영속성 컨텍스트에서 PK가 같은 객체는 무조건 같아야 한다
원래대로라면 refMember는 프록시, findMember는 원래 진짜 멤버 클래스여야 하는데

refMember와 findMember가 같아야 하기 때문에
em.find()로 가지고 온 객체도 프록시 클래스 타입이 된다.


```

Member refMember = em.getReference(Member.class, member1.getId());
System.out.println("refMember = " + refMember.getClass()); //Proxy

em.detach(refMember);

System.out.println("refMember = " + refMember.getUsername());

tx.commit();
} catch (Exception e) {
    tx.rollback();
    System.out.println("e = " + e);
}

```

```

refMember = class hellojpa.Member$HibernateProxy$0luamfNY
org.hibernate.LazyInitializationException: could not initialize proxy [hellojpa.Member#1] - no Session <4
    at hellojpa.Member$HibernateProxy$0luamfNY.getUsername(Unknown Source)
    at hellojpa.JpaMain.main(JpaMain.java:32)
Jun 08, 2019 11:35:55 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImp
INFO: HHH10001008: Cleaning up connection pool [jdbc:h2:tcp://localhost/~/test]

```

em.detach(refMember) or em.close() or em.clear() 한 다음에
 refMember.getUsername() 을 호출
 프록시를 초기화 할 수 없다 - 세션이 없다 (영속성 컨텍스트에 없다)는 에러가 난다

프록시 확인

- 프록시 인스턴스의 초기화 여부 확인

PersistenceUnitUtil.isLoaded(Object entity)
emf.getPersistenceUnitUtil.isLoaded(refMember)

- 프록시 클래스 확인 방법

entity.getClass().getName() 출력(..javasist.. or
HibernateProxy...)

- 프록시 강제 초기화

org.hibernate.Hibernate.initialize(entity);

- 참고: JPA 표준은 강제 초기화 없음

강제 호출: **member.getName()** 이런걸로 강제 초기화해야 함

즉시 로딩과 지연 로딩

Member를 조회할 때 Team도 함께 조회해야 할까?

단순히 member 정보만 사용하는 비즈니스 로직
`println(member.getName());`



지연 로딩 LAZY을 사용해서 프록시로 조회

```
@Entity
public class Member {

    @Id
    @GeneratedValue
    private Long id;

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

    @ManyToOne(fetch = FetchType.LAZY) /**
    @JoinColumn(name = "TEAM_ID")
    private Team team;
    ..
}
```

```
Member m = em.find(Member.class, member1.getId());  
  
System.out.println("m = " + m.getTeam().getClass());  
  
System.out.println("=====");  
m.getTeam().getName();  
System.out.println("=====");
```

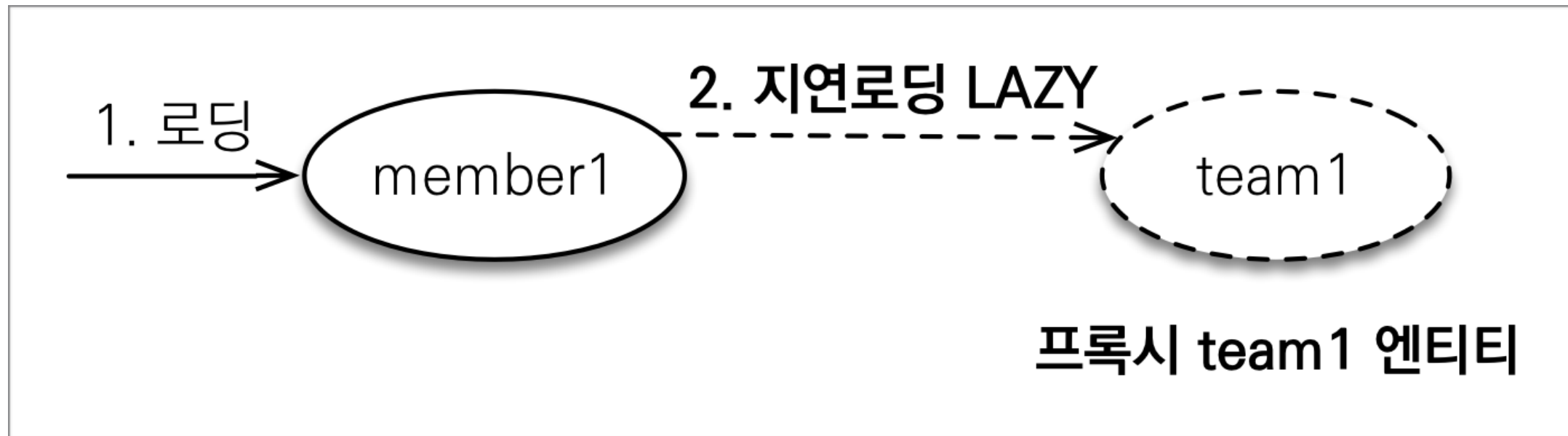
em.find() 로 Member 조회 시 member 에 대해서만 select 쿼리 날아감

m.getTeam().getClass() 로 출력해보면 지연로딩으로 설정된 팀은
프록시 객체로 가져왔음을 알 수 있음.

m.getTeam().getName();

프록시.getName() 을 해야 비로소 select 쿼리가 날아간다

지연 로딩



지연 로딩 LAZY을 사용해서 프록시로 조회



```
Member member = em.find(Member.class, 1L);
```



```
Team team = member.getTeam();  
team.getName(); // 실제 team을 사용하는 시점에 초기화(DB 조회)
```

Member와 Team을 자주 함께 사용한다면?



즉시 로딩 EAGER를 사용해서 함께 조회

```
@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;
    ..
}
```


Hibernate:

select

```
member0_.MEMBER_ID as MEMBER_I1_3_0_,  
member0_.createdBy as createdB2_3_0_,  
member0_.createdDate as createdD3_3_0_,  
member0_.lastModifiedBy as lastModi4_3_0_,  
member0_.lastModifiedDate as lastModi5_3_0_,  
member0_.team_TEAM_ID as team_TEA7_3_0_,  
member0_.USERNAME as USERNAME6_3_0_,  
team1_.TEAM_ID as TEAM_ID1_7_1_,  
team1_.createdBy as createdB2_7_1_,  
team1_.createdDate as createdD3_7_1_,  
team1_.lastModifiedBy as lastModi4_7_1_,  
team1_.lastModifiedDate as lastModi5_7_1_,  
team1_.name as name6_7_1_
```

from

Member member0_

left outer join

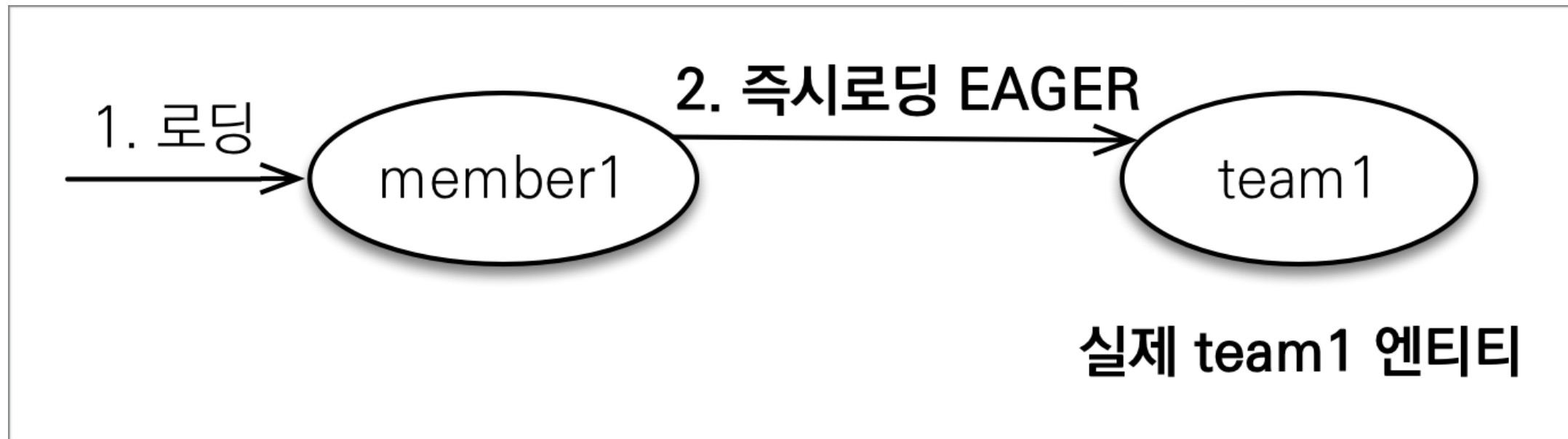
Team team1_

on member0_.team_TEAM_ID=team1_.TEAM_ID |

where

member0_.MEMBER_ID=?

즉시 로딩



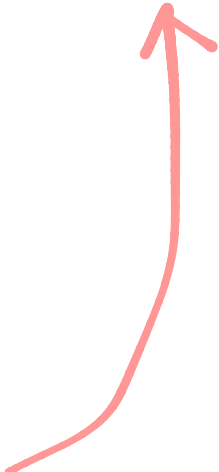
즉시 로딩(EAGER), Member조회시 항상 Team도 조회



JPA 구현체는 가능하면 조인을 사용해서 SQL 한번에 함께 조회

프록시와 즉시로딩 주의

우선 Lazy Loading 으로 바른 다음에
1. Fetch join (동적으로 조인을 이용)
2. 엔티티 그래프
3. 배치 사이즈? 등의 방법으로 해결하면 된다.

- **가급적 지연 로딩만 사용(특히 실무에서)**
Lazy Loading
 - 즉시 로딩을 적용하면 예상하지 못한 SQL이 발생
 - **즉시 로딩은 JPQL에서 N+1 문제를 일으킨다.**
처음 쿼리를 하나 날렸는데, 그 때 조회 해온 데이터 개수(N) 만큼 또 쿼리를 날리는 문제
 - **@ManyToOne, @OneToOne은 기본이 즉시 로딩**
-> LAZY로 설정
 - @OneToMany, @ManyToMany는 기본이 지연 로딩
- 

즉시로딩은 JPQL에서 N+1 문제를 일으킨다

예를들어

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

이 때 Member 의 Team 필드가 EAGER 라면 쿼리가 1+(멤버의 수) 만큼 나간다

em.find() 는 JPA 가 내부적으로 최적화 해줬지만

JPQL 은 그대로가 sql 로 번역된다. (Select * from Member;)

쿼리를 보내서 멤버를 가지고 왔더니 Team 이 즉시로딩이네?

그러면 멤버 쿼리 나가고, 멤버의 수 만큼 팀을 조회해오는 쿼리가 또 날아가는 것
(LAZY면 그냥 프록시를 집어 넣었을 거라서 sql 쿼리가 날아가진 않았을 것)

따라서 JPQL 을 쓸 때는 지연 로딩을 사용하는 게 좋다.

@OneToOne 의 경우 지연로딩이 잘 작동하지 않을 수 있다.

<https://yongkyu-jang.medium.com/jpa-%EB%8F%84%EC%9E%85-onetoone-%EA%B4%80%EA%B3%84%EC%97%90%EC%84%9C%EC%9D%98-lazyloading-%EC%9D%B4%EC%8A%88-1-6d19edf5f4d3>

JPA에서의 OneToOne 관계에서의 Lazy Loading 발동조건

1. nullable이 허용되지 않는 1:1 관계. 즉, 참조 객체가 optional = false 로 지정할 수 있는 관계여야 한다.
2. 양방향인 아닌 단방향 1:1 관계여야 한다.
3. @PrimaryKeyJoin은 허용되지 않는다. 부모와 자식 엔티티간의 조인컬럼이 모두 PK의 경우를 의미한다.

얼핏 보면 특별해 보이지 않는 조건인 것 같다. 보통의 1:1 관계에서는 자식엔티티가 Null을 허용하는 경우가 필수인 경우보다는 많을 것이다. 또한, 대부분의 테이블간의 관계는 외래키 조인을 통해서 설정하는게 일반적인 방식이라고 보면 위의 조건중 1번과 3번을 만족시키는게 어렵지는 않다.

예제의 경우, 1,2번은 해당사항이 아니다.

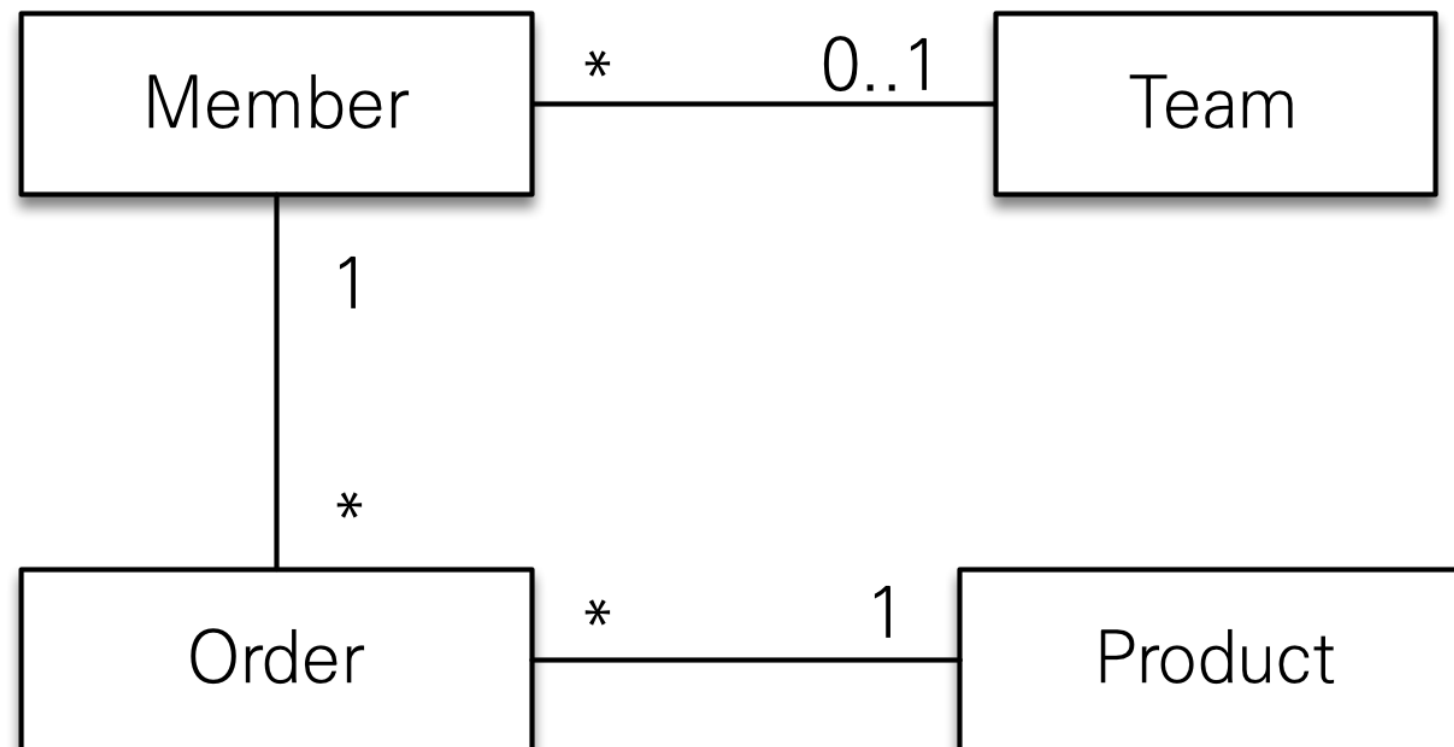
문제는 2번이다. 1:1에서는 양방향 관계가 설정되면 LazyLoading이 발동하지 않는다.

지연 로딩 활용

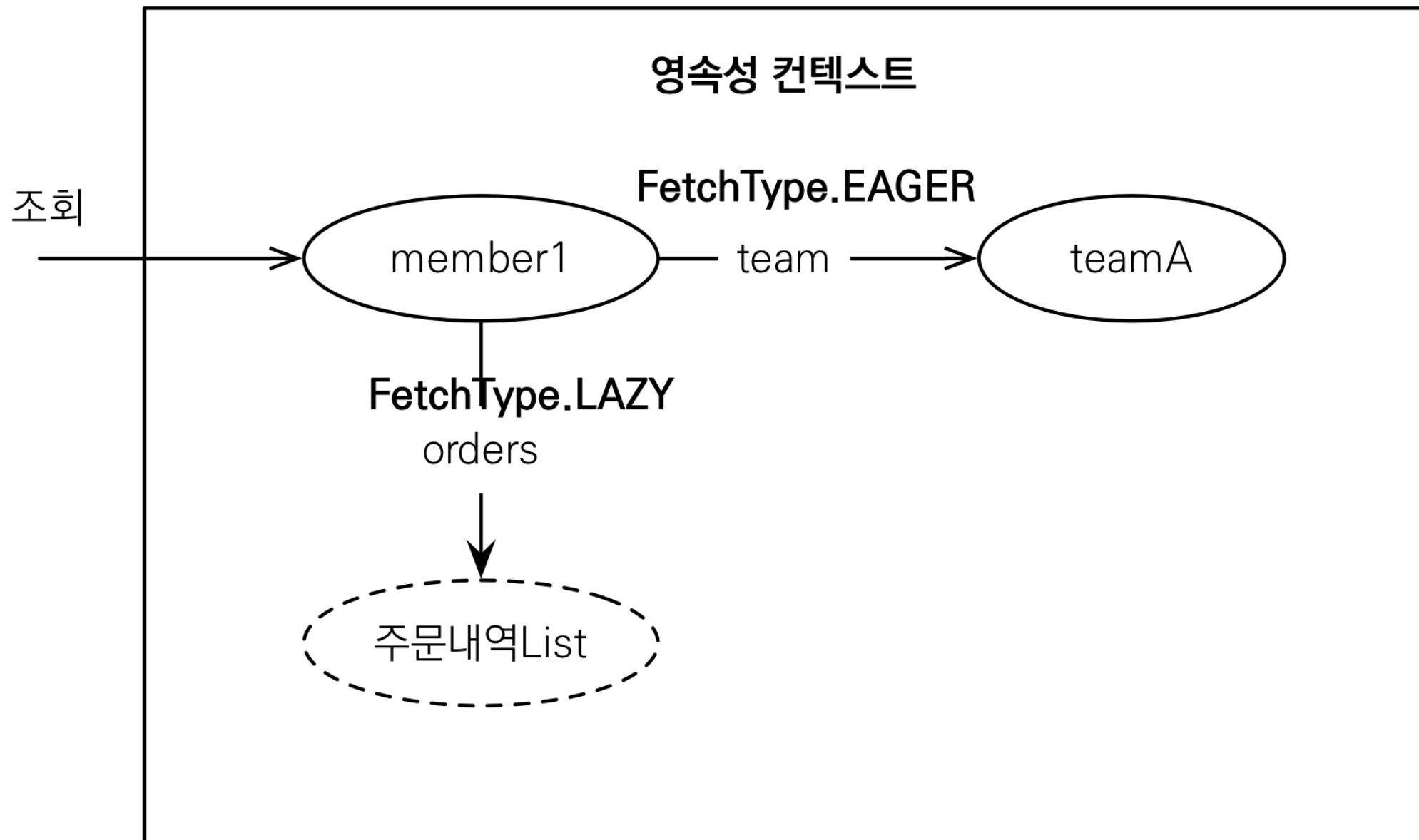
지연 로딩 활용

이론적으로 한번 알아보자는 거지
실무에서는 모두 지연 로딩으로 발라야 한다!!

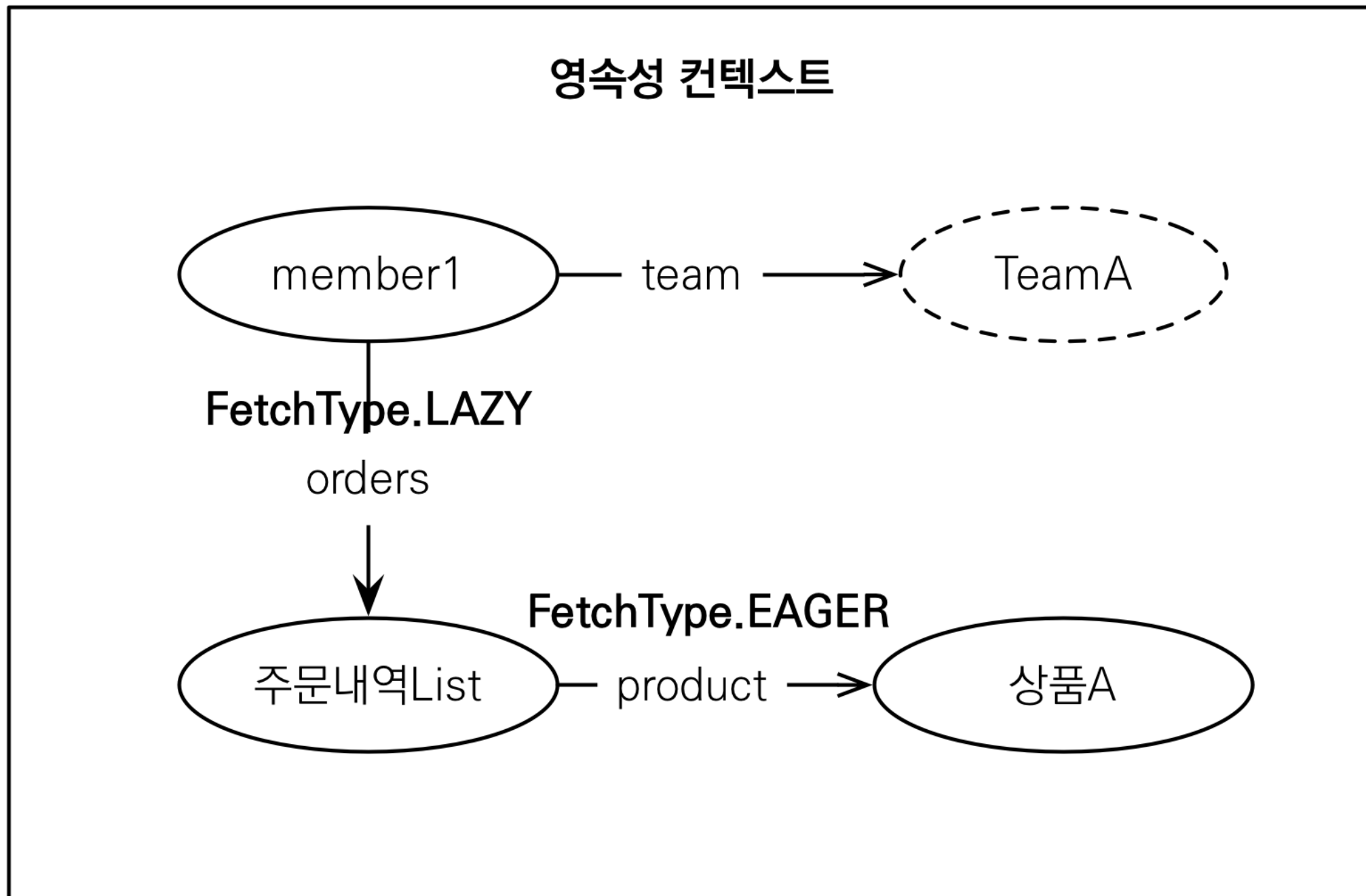
- **Member**와 **Team**은 자주 함께 사용 -> 즉시 로딩
- **Member**와 **Order**는 가끔 사용 -> 지연 로딩
- **Order**와 **Product**는 자주 함께 사용 -> 즉시 로딩



지연 로딩 활용



지연 로딩 활용



지연 로딩 활용 - 실무

- 모든 연관관계에 지연 로딩을 사용해라!
- 실무에서 즉시 로딩을 사용하지 마라!
- JPQL fetch 조인이나, 엔티티 그래프 기능을 사용해라!
(뒤에서 설명)
- 즉시 로딩은 상상하지 못한 쿼리가 나간다.

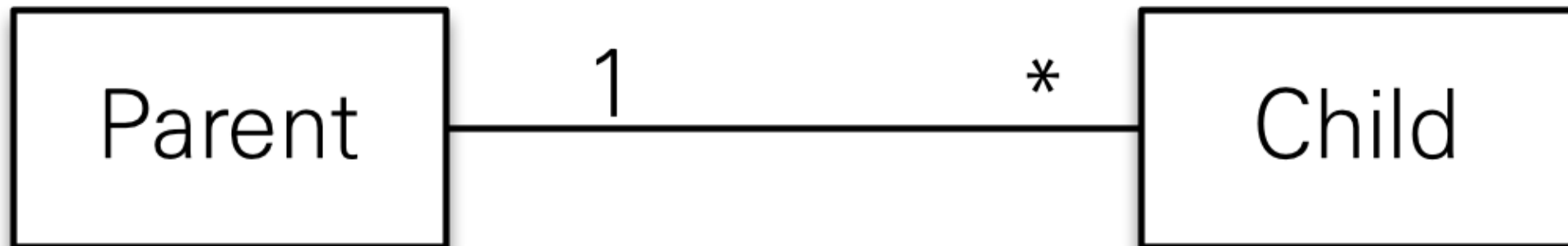
영속성 전이: CASCADE

즉시로딩 지연로딩과 아무 상관 없다!!!

게시판에서 글과 첨부파일에 관한 데이터 정도는 cascade 로 묶어도 되는데
(소유자가 하나일 때, 부모와 자식의 라이프사이클 - 등록, 삭제 가 동일할 때)
Child 를 다른 테이블에서도 쓰고 있는 경우는 cascade 를 쓰지 않아야 할 때가 많다.

영속성 전이: CASCADE

- 특정 엔티티를 영속 상태로 만들 때 연관된 엔티티도 함께 영속 상태로 만들도 싶을 때
- 예: 부모 엔티티를 저장할 때 자식 엔티티도 함께 저장.



@Entity

```
public class Parent {  
    @Id @GeneratedValue  
    private Long id;  
  
    private String name;  
  
    @OneToMany(mappedBy = "parent")  
    private List<Child> childList = new ...  
  
    // 편의 메소드  
    public void addChild(Child child) {  
        childList.add(child);  
        child.setParent(this);  
    }  
}
```

@Entity

```
public class Child {  
    @Id @GeneratedValue  
    private Long id;  
  
    private String name;  
  
    @ManyToOne  
    @JoinColumn(name = "parent_id")  
    private Parent parent;  
}
```

Child child1, child2 = new ...

Parent parent = new ...

parent.addChild(child1);

parent.addChild(child2);

em.persist(parent)

em.persist(child1)

em.persist(child2)

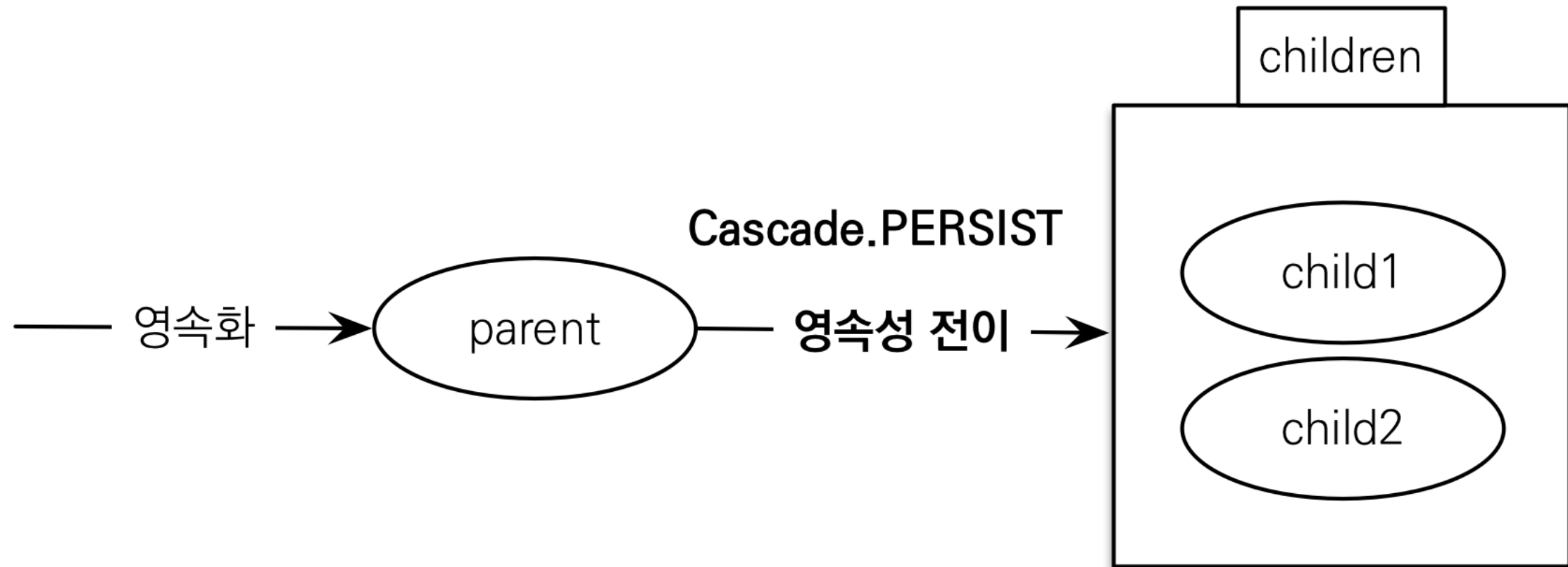
Persist를 3번 호출해야 insert 가 된다.

이럴때 cascade 를 이용하면 편리하다

Parent 가 persist 될 때 Child 도 persist 해준다
(Persist 여야 DB 에 저장이 된다)

영속성 전이: 저장

```
@OneToMany(mappedBy="parent", cascade=CascadeType.PERSIST)
```



영속성 전이: CASCADE - 주의!

- 영속성 전이는 연관관계를 매핑하는 것과 아무 관련이 없음
- 엔티티를 영속화할 때 연관된 엔티티도 함께 영속화하는 편리함을 제공할 뿐

CASCADE의 종류

- **ALL: 모두 적용** - 삭제까지 전이
- **PERSIST: 영속** - 부모가 저장될 때 자식도 같이 저장하려고 하는 경우 (삭제는 전이 되지 않음)
- **REMOVE: 삭제**
- **MERGE: 병합**
- **REFRESH: REFRESH**
- **DETACH: DETACH**

고아 객체

고아 객체

- 고아 객체 제거: 부모 엔티티와 연관관계가 끊어진 자식 엔티티를 자동으로 삭제
- **orphanRemoval = true**
- `Parent parent1 = em.find(Parent.class, id);`
`parent1.getChildren().remove(0);` 연관 관계를 끊는다.
//자식 엔티티를 컬렉션에서 제거 (힙에서 자식 객체를 지우는게 아님)
- `DELETE FROM CHILD WHERE ID=?`
이후 `parent1` 를 `persist` 하면 0번째에 있었던 `child` 는 DB 에서 지워져야 하는가?

```
@OneToMany(mappedBy = "parent", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Child> childList = new ArrayList<>();
```

고아 객체 - 주의

- 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 ^{em.remove(parent)} 부모를 제거하면 ^{parent.getChildren()} 자식은 고아가 된다. 따라서 고아 객체 제거 기능을 활성화 하면, 부모를 제거할 때 자식도 함께 제거된다. 이것은 CascadeType.REMOVE처럼 동작한다.

영속성 전이 + 고아 객체, 생명주기

영속성 전이 + 고아 객체, 생명주기

- **CascadeType.ALL + orphanRemovel=true**
- 스스로 생명주기를 관리하는 엔티티는 em.persist()로 영속화, em.remove()로 제거
- 두 옵션을 모두 활성화 하면 부모 엔티티를 통해서 자식의 생명 주기를 관리할 수 있음
Child 는 repository 도 없어도 된다. 그냥 부모를 저장하면 같이 저장하고, 부모를 지우면 같이 지워짐
- 도메인 주도 설계(DDD)의 Aggregate Root개념을 구현할 때 유용

실전 예제 - 5.연관관계 관리

글로벌 페치 전략 설정

- 모든 연관관계를 지연 로딩으로
- @ManyToOne, @OneToOne은 기본이 즉시 로딩이므로 지연 로딩으로 변경

영속성 전이 설정

- **Order -> Delivery**를 영속성 전이 ALL 설정
- **Order -> OrderItem**을 영속성 전이 ALL 설정