- **심화 주제**
  - **예외 처리**
    - - JPA 표준 예외 정리
    - - 스프링 프레임워크의 JPA 예외 변환
    - - 스프링 프레임워크에 JPA 예외 변환기 적용하기
    - - 트랜잭션 롤백시 주의사항
  - **엔티티 비교하기**
    - - 영속성 컨텍스트가 같을 때 엔티티 비교하기
    - - 영속성 컨텍스트가 다를 때 엔티티 비교하기
  - **프록시 심화 주제**
    - - 영속성 컨텍스트와 프록시
    - - 프록시 타입 비교
    - - 프록시 동등성 비교
    - - 상속관계와 프록시
      - – JPQL로 대상 직접 조회하기
      - – 프록시 벗기기
      - – 기능을 위한 별도의 인터페이스 제공
      - – 비지터 패턴 사용

심화 주제

JPA의 깊이 있는 심화 주제들을 알아보자.

- 예외 처리: JPA를 사용할 때 발생하는 다양한 예외와 예외에 따른 주의점을 설명한다.

- 엔티티 비교하기: 엔티티를 비교할 때 주의점과 해결 방법을 설명한다.
- 프록시 심화 주제: 프록시로 인해 발생하는 다양한 문제점과 해결 방법을 다룬다.

예외 처리

- JPA 표준 예외 정리

JPA 표준 예외들은 `javax.persistence.PersistenceException`의 자식 클래스다. 그리고 이 예외 클래스는 `RuntimeException`의 자식이다.

JPA 표준 예외는 크게 2가지로 나눌 수 있다.

- 트랜잭션 롤백을 표시하는 예외
- 트랜잭션 롤백을 표시하지 않는 예외

트랜잭션 롤백을 표시하는 예외는 심각한 예외이므로 복구해선 안 된다. 이 예외가 발생하면 트랜잭션을 강제로 커밋해도 트랜잭션이 커밋되지 않고 대신에 `javax.persistence.RollbackException` 예외가 발생한다. 반면에 트랜잭션 롤백을 표시하지 않는 예외는 심각한 예외가 아니다. 따라서 개발자가 트랜잭션을 커밋할지 롤백할지를 판단하면 된다.

트랜잭션 롤백을 표시하는 예외

트랜잭션 롤백을 표시하는 예외	설명
`javax.persistence.EntityExistsException`	`EntityManager.persist(..)` 호출 이미 같은 엔티티가 있으면 발생한다.
`javax.persistence.EntityNotFoundException`	`EntityManager.getReference(..` 호출했는데 실제 사용시 엔티티가 존재 지 않으면 발생, `refresh(..)` , `lock(.. )` 에서도 발생한다.
`javax.persistence.OptimisticLockException`	낙관적 락 충돌시 발생한다.
`javax.persistence.PessimisticLockException`	비관적 락 충돌시 발생한다.
`javax.persistence.RollbackException`	`EntityTransaction.commit()` 실 시 발생, 롤백이 표시되어 있는 트랜잭 커밋시에도 발생한다.
`javax.persistence.TransactionRequiredException`	트랜잭션이 필요할 때 트랜잭션이 없 발생, 트랜잭션 없이 엔티티를 변경할 주로 발생한다.

트랜잭션 롤백을 표시하지 않는 예외

트랜잭션 롤백을 표시하지 않는 예외	설명
`javax.persistence.NoResultException`	`Query.getSingleResult()` 호출시 결과가 하나도 없을 때 발생한다.
`javax.persistence.NonUniqueResultException`	`Query.getSingleResult()` 호출시 결과가 둘 이상일 때 발생한다.
`javax.persistence.LockTimeoutException`	비관적 락에서 시간 초과시 발생한다.
`javax.persistence.QueryTimeoutException`	쿼리 실행 시간 초과시 발생한다.

- 스프링 프레임워크의 JPA 예외 변환

서비스 계층에서 데이터 접근 계층의 구현 기술에 직접 의존하는 것은 좋은 설계라 할 수 없다. 이것은 예외도 마찬가지인데, 예를 들어 서비스 계층에서 JPA의 예외를 직접 사용하면 JPA에 의존하게 된다.

스프링 프레임워크는 이런 문제를 해결하려고 데이터 접근 계층에 대한 예외를 추상화해서 개발자에게 제공한다.

JPA 예외를 스프링 예외로 변경

JPA 예외	스프링 변환 예외
`javax.persistence.PersistenceException`	`org.springframework.orm.jpa.J`
`javax.persistence.NoResultException`	`org.springframework.dao.Empty`
`javax.persistence.NonUniqueResultException`	`org.springframework.dao.Incor`
`javax.persistence.LockTimeoutException`	`org.springframework.dao.Canno`
`javax.persistence.QueryTimeoutException`	`org.springframework.dao.Query`
`javax.persistence.EntityExistsException`	`org.springframework.dao.DataI`
`javax.persistence.EntityNotFoundException`	`org.springframework.orm.jpa.J`
`javax.persistence.OptimisticLockException`	`org.springframework.orm.jpa.J`
`javax.persistence.PessimisticLockException`	`org.springframework.dao.Pessi`
`javax.persistence.TransactionRequiredException`	`org.springframework.dao.InvaI`
`javax.persistence.RollbackException`	`org.springframework.transacti`

추가로 JPA 표준 명세상 발생할 수 있는 다음 두 예외도 추상화해서 제공한다.

JPA 예외	스프링 변환 예외
`java.lang.IllegalStateException`	`org.springframework.dao.InvalidDataAccess`
`java.lang.IllegalArgumentException`	`org.springframework.dao.InvalidDataAccess`

- 스프링 프레임워크에 JPA 예외 변환기 적용하기

JPA 예외를 스프링 프레임워크가 제공하는 추상화된 예외로 변경하려면 `PersistenceExceptionTranslationPostProcessor`를 스프링 빈으로 등록하면 된다. 이것은 `@Repository` 어노테이션을 사용한 곳에 예외 변환 AOP를 적용해서 JPA 예외를 스프링 프레임워크가 추상화한 예외로 변환해준다. 설정 방법은 다음과 같다.

====== JavaConfig ======

```
@Bean
public PersistenceExceptionTranslationPostProcessor exceptionTranslation() {
    return new PersistenceExceptionTranslationPostProcessor();
}
```

====== XML ======

```
<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationP
```

예외 변환 예제 코드를 보자.

====== 예외 변환 예제 코드 ======

```
@Repository //**
public class NoResultExceptionTestRepository {

    @PersistenceContext EntityManager em;

    public void findMember() {
        //조회된 데이터가 없음
        em.createQuery("select m from Member m", Member.class)
            .getSingleResult(); //**
    }
}
```

`findMember()` 메서드는 엔티티를 조회하려고 `getSingleResult()` 메서드를 사용했다. 이 메서드는 조회된 결과가 없으면 `javax.persistence.NoResultException`이 발생한다. 이 예외가 `findMember()` 메서드를 빠져 나갈 때 `PersistenceExceptionTranslationPostProcessor`에서 등록한 AOP 인터셉터가 동작해서 해당 예외를 `org.springframework.dao.EmptyResultDataAccessException` 예외로 변환해서 반환한다. 따라서 이 메서드를 호출한 클라이언트는 스프링 프레임워크가 추상화한 예외를 받는다.

만약 예외를 변환하지 않고 그대로 반환하고 싶으면 다음처럼 `throws` 절에 그대로 반환할 JPA 예외나 JPA 예외의 부모 클래스를 직접 명시하면 된다. 예를 들어 `java.lang.Exception`를 선언하면 모든 예외의 부모이므로 예외를 변환하지 않는다.

====== 예외를 변환하지 않는 코드 ======

```
@Repository
public class NoResultExceptionTestService {

    @PersistenceContext EntityManager em;

    public void findMember() throws javax.persistence.NoResultException {
        em.createQuery("select m from Member m", Member.class).getSingleResult()
    }
}
```

- 트랜잭션 롤백시 주의사항

트랜잭션을 롤백하는 것은 데이터베이스의 반영사항만 롤백하는 것이지 수정한 자바 객체까지 원상태로 복구해 주지는 않는다. 예를 들어 엔티티를 조회해서 수정하는 중에 문제가 있어서 트랜잭션을 롤백하면 데이터베이스의 데이터는 원래대로 복구 되지만 객체는 수정된 상태로 영속성 컨텍스트에 남아있다. 따라서 트랜잭션이 롤백된 영속성 컨텍스트를 그대로 사용하는 것은 위험하다. 새로운 영속성 컨텍스트를 생성해서 사용하거나 `EntityManager.clear()`를 호출해서 영속성 컨텍스트를 초기화한 다음에 사용해야 한다.

스프링 프레임워크는 이런 문제를 예방하기 위해 영속성 컨텍스트의 범위에 따라 다른 방법을 사용한다.

기본 전략인 트랜잭션당 영속성 컨텍스트 전략은 문제가 발생하면 트랜잭션 AOP 종료 시점에 트랜잭션을 롤백하면서 영속성 컨텍스트도 함께 종료하므로 문제가 발생하지 않는다.

문제는 OSIV처럼 영속성 컨텍스트의 범위를 트랜잭션 범위보다 넓게 사용해서 여러 트랜잭션이 하나의 영속성 컨텍스트를 사용할 때 발생한다. 이때는 트랜잭션을 롤백해서 영속성 컨텍스트에 이상이 발생해도 다른 트랜잭션에서 해당 영속성 컨텍스트를 그대로 사용하는 문제가 있다. 스프링 프레임워크는 영속성 컨텍스트의 범위를 트랜잭션의 범위보다 넓게 설정하면 트랜잭션 롤백시 영속성 컨텍스트를 초기화(`EntityManager.clear()`)해서 잘못된 영속성 컨텍스트를 사용하는 문제를 예방한다.

더 자세한 내용은 `org.springframework.orm.jpa.JpaTransactionManager`의 `doRollback()` 메서드를 참고하자.

엔티티 비교하기

영속성 컨텍스트 내부에는 엔티티 인스턴스를 보관하기 위한 1차 캐시가 있다. 이 1차 캐시는 영속성 컨텍스트와 생명주기를 같이 한다.

영속성 컨텍스트를 통해 데이터를 저장하거나 조회하면 1차 캐시에 엔티티가 저장된다. 이 1차 캐시 덕분에 변경 감지 기능도 동작하고, 이름 그대로 1차 캐시로 사용되어서 데이터베이스를 통하지 않고 데이터를 바로 조회할 수도 있다.

애플리케이션 수준의 반복 가능한 읽기

영속성 컨텍스트를 더 정확히 이해하기 위해서는 1차 캐시의 가장 큰 장점인 **반복 가능한 읽기**를 이해해야 한다. 영속성 컨텍스트에서 엔티티를 조회하면 항상 같은 엔티티 인스턴스를 반환한다. 이것은 단순히 동등성(equals) 비교 수준이 아니라 정말 주소값이 같은 인스턴스를 반환한다.

```
Member member1 = em.find(Member.class, "1L");
Member member2 = em.find(Member.class, "1L");

assertTrue(member1 == member2); //둘은 같은 인스턴스다.
```

- 영속성 컨텍스트가 같을 때 엔티티 비교하기

이전에 작성했던 회원가입 테스트 케이스로 트랜잭션 범위의 영속성 컨텍스트와 반복 가능한 읽기에 대해 더 자세히 알아보자.

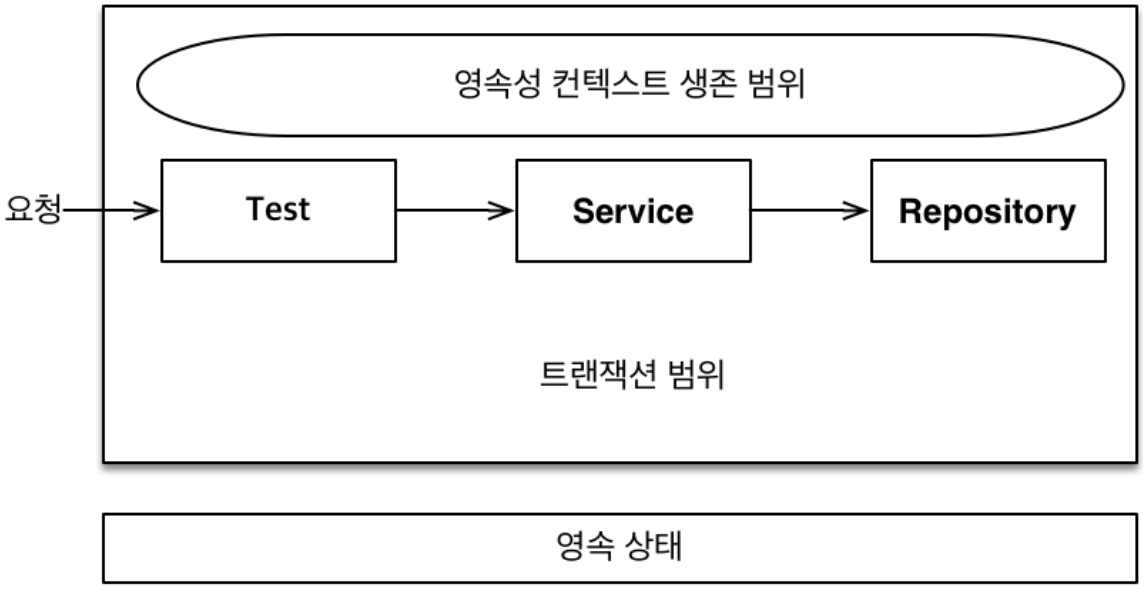

그림 - 테스트와 트랜잭션 범위

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:appConfig.xml")
@Transactional //트랜잭션 안에서 테스트를 실행한다.
public class MemberServiceTest {

    @Autowired MemberService memberService;
    @Autowired MemberRepository memberRepository;

    @Test
    public void 회원가입() throws Exception {

        //Given
        Member member = new Member("kim");

        //When
        Long saveId = memberService.join(member);

        //Then
        Member findMember = memberRepository.fineOne(saveId)
        assertTrue(member == findMember); //참조값 비교
    }
}

@Transactional
public class MemberService {

    @Autowired MemberRepository memberRepository;

    public Long join(Member member) {
        ...
```

```
            memberRepository.save(member);
            return member.getId();
        }
    }

    @Repository
    public class MemberRepository {

        @PersistenceContext
        EntityManager em;

        public void save(Member member) {
            em.persist(member);
        }

        public Member findOne(Long id) {
            return em.find(Member.class, id);
        }
    }
```

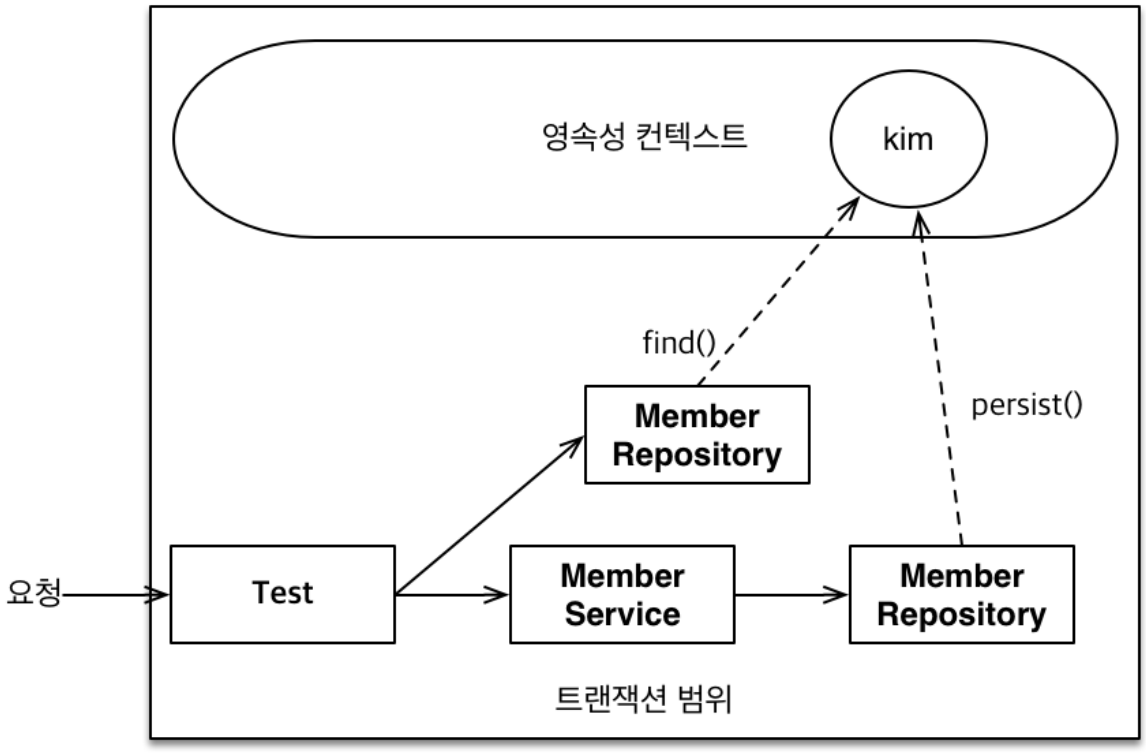

그림 - 테스트와 트랜잭션 범위 예제

테스트 클래스에 `@Transactional`이 선언되어 있으면 트랜잭션을 먼저 시작하고 테스트 메서드를 실행한다. 따라서 테스트 메서드인 `회원가입()`은 이미 트랜잭션 범위에 들어있고 이 메서드가 끝나면 트랜잭션이 종료된다. 그러므로 `회원가입()`에서 사용된 코드는 항상 같은 트랜잭션과 같은 영속성 컨텍스트에 접근한다.

코드를 보면 회원을 생성하고 `memberRepository`에서 `em.persist(member)`로 회원을 영속성 컨텍스트에 저장한다. 그리고 저장된 회원을 찾아서 저장한 회원과 비교한다.

```
Member findMember = memberRepository.fineOne(saveId)
assertTrue(member == findMember); //참조값 비교
```

여기서 흥미로운 것은 저장한 회원과 회원 리파지토리에서 찾아온 엔티티가 완전히 같은 인스턴스라는 점이다. 이것은 같은 트랜잭션 범위에 있으므로 같은 영속성 컨텍스트를 사용하기 때문이다.

따라서 영속성 컨텍스트가 같으면 다음 3가지 조건을 모두 만족한다.

- 동일성(identical): `==` 비교가 같다.
- 동등성(equinalent): `equals()` 비교가 같다.
- 데이터베이스 동등성: `@Id`인 데이터베이스 식별자가 같다.

> **참고:** 테스트에도 `@Transactional`이 있고 서비스에도 `@Transactional`이 있다. 기본 전략은 먼저 시작된 트랜잭션이 있으면 그 트랜잭션을 그대로 이어 받아 사용하고 없으면 새로 시작한다. 만약 다른 전략을 사용하고 싶으면 `propagation`속성을 변경하면 된다. 자세한 내용은 스프링 문서를 읽어보자.
>
> ```
> @Transactional(propagation = Propagation.REQUIRED) //기본값
> ```

> **참고:** 테스트 클래스에 `@Transactional`을 적용하면 테스트가 끝날때 트랜잭션을 커밋하지 않고 트랜잭션을 강제로 롤백한다. 그래야 데이터베이스에 영향을 주지 않고 테스트를 반복해서 할 수 있기 때문이다. 문제는 롤백시에는 영속성 컨텍스트를 플러시하지 않는다는 점이다. 플러시를 하지 않으므로 플러시 시점에 어떤 SQL이 실행되는지 콘솔 로그에 남지 않는다. 어떤 SQL이 실행되는지 콘솔을 통해 보고 싶으면 테스트 마지막에 `em.flush()`를 강제로 호출하면 된다.

- 영속성 컨텍스트가 다를 때 엔티티 비교하기

테스트 클래스에 `@Transactional`이 없고 서비스에만 `@Transactional`이 있으면 다음 그림과 같은 트랜잭션 범위와 영속성 컨텍스트 범위를 가지게 된다.

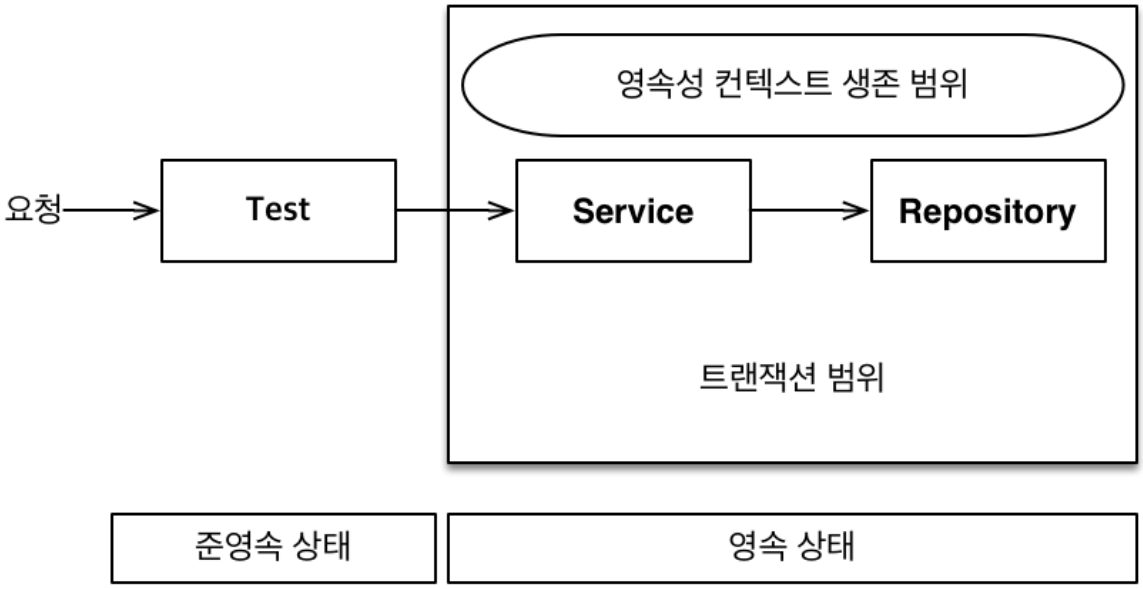

그림 - 테스트와 트랜잭션 - 준영속

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:appConfig.xml")
//@Transactional //테스트에서 트랜잭션을 사용하지 않는다. //**
public class MemberServiceTest {

    @Autowired MemberService memberService;
    @Autowired MemberRepository memberRepository;

    @Test
    public void 회원가입() throws Exception {

        //Given
        Member member = new Member("kim");

        //When
        Long saveId = memberService.join(member);

        //Then
        Member findMember = memberRepository.fineOne(saveId)
        //findMember는 준영속 상태다.
        assertTrue(member == findMember); //둘은 다른 주소값을 가진 인스턴스다. 테스트가 실
    }
}

@Transactional //서비스 클래스에서 트랜잭션이 시작된다. //**
public class MemberService {

    @Autowired MemberRepository memberRepository;

    public Long join(Member member) {
```

```
            ...
            memberRepository.save(member);
            return member.getId();
        }
    }

    @Repository
    @Transactional //예제를 구성하기 위해 추가했다. //**
    public class MemberRepository {

        @PersistenceContext
        EntityManager em;

        public void save(Member member) {
            em.persist(member);
        }

        public Member findOne(Long id) {
            return em.find(Member.class, id);
        }
    }
```

이 테스트는 실패하는데 왜 실패하는지 이해했다면 영속성 컨텍스트의 생존 범위에 대해 거의 이해했다고 볼 수 있다. 예제를 구성하기 위해 `MemberRepository`에도 `@Transactional`을 추가했다.

그림을 보면서 설명하겠다.

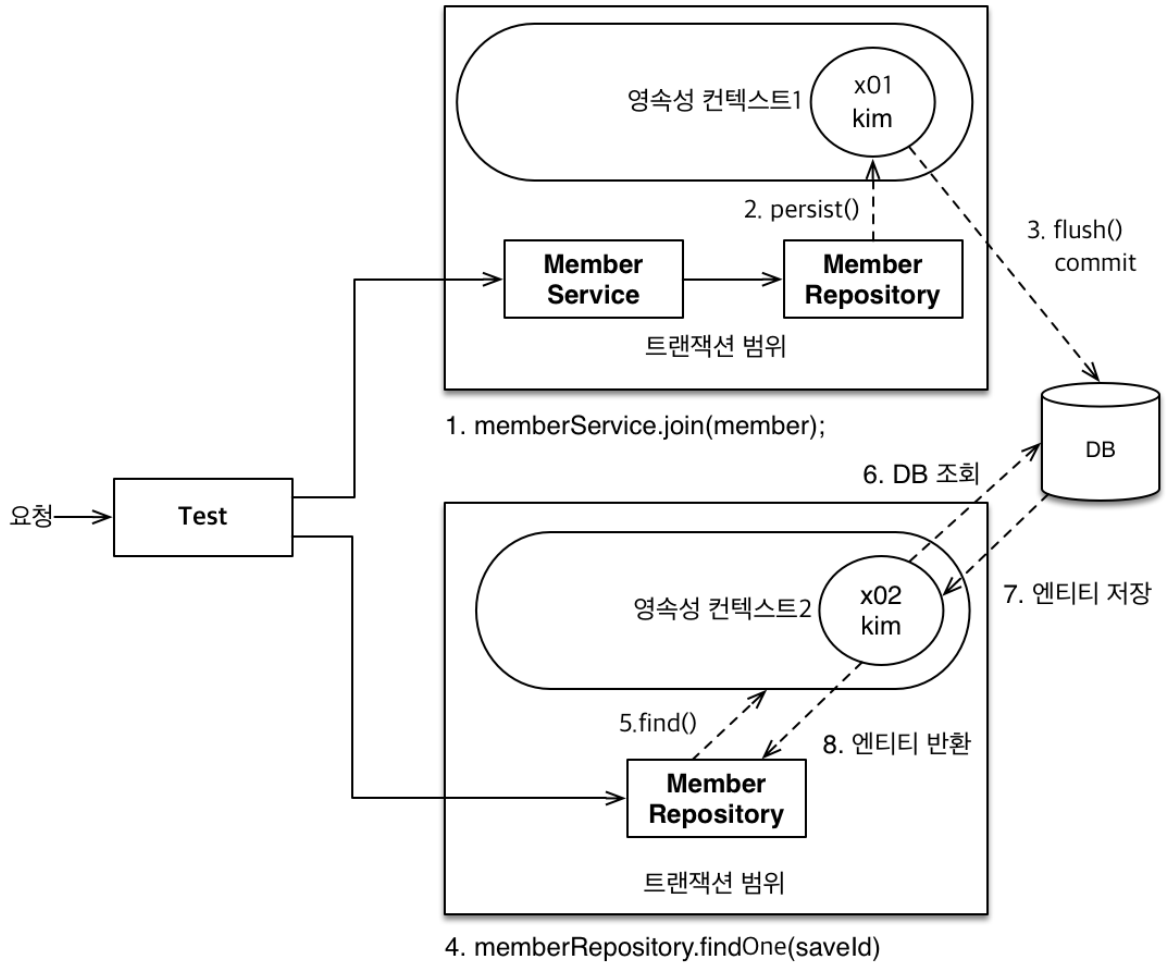

그림 - 테스트와 트랜잭션 - 준영속 예제

1. 테스트 코드에서 `memberService.join(member)` 을 호출해서 회원가입을 시도하면 서비스 계층에서 트랜잭션이 시작되고 **영속성 컨텍스트1**이 만들어진다.
2. `memberRepository`에서 `em.persist()`를 호출해서 `member` 엔티티를 영속화한다.
3. 서비스 계층이 끝날 때 트랜잭션이 커밋되면서 영속성 컨텍스트가 플러시 된다. 이때 트랜잭션과 영속성 컨텍스트가 종료된다. 따라서 `member` 엔티티 인스턴스는 준영속 상태가 된다.
4. 테스트 코드에서 `memberRepository.findOne(saveId)` 을 호출해서 저장한 엔티티를 조회하면 리파지토리 계층에서 새로운 트랜잭션이 시작되면서 새로운 **영속성 컨택스트2**가 생성된다.
5. 저장된 회원을 조회하지만 새로 생성된 영속성 컨텍스트2에는 찾는 회원이 존재하지 않는다.
6. 따라서 데이터베이스에서 회원을 찾아온다.
7. 데이터베이스에서 조회된 회원 엔티티를 영속성 컨텍스트에 보관하고 반환한다.
8. `memberRepository.fineOne()` 메서드가 끝나면서 트랜잭션이 종료되고 영속성 컨텍스트2도 종료된다.

`member`와 `findMember`는 각각 다른 영속성 컨텍스트에서 관리되었기 때문에 둘은 다른 인스턴스다.

```
assertTrue(member == findMember) //실패한다.
```

하지만 `member`와 `findMember`는 인스턴스는 다른지만 같은 데이터베이스 로우를 가르키고 있다. 따라서 사실상 같은 엔티티로 보아야 한다.

이처럼 영속성 컨텍스트가 다르면 동일성 비교가 실패한다.

- 동일성(identical): `==` 비교가 실패한다.
- 동등성(equinalent): `equals()` 비교가 만족한다. 단 `equals()`를 구현해야 한다. 보통 비즈니스 키로 구현한다.
- 데이터베이스 동등성: `@Id`인 데이터베이스 식별자가 같다.

앞서 보았듯이 같은 영속성 컨텍스트를 보장하면 동일성 비교만으로 충분하다. 따라서 OSIV처럼 요청의 시작부터 끝까지 같은 영속성 컨텍스트를 사용할 때는 동일성 비교가 성공한다. 하지만 지금처럼 영속성 컨텍스트가 달라지면 동일성 비교는 실패한다. 따라서 엔티티의 비교에 다른 방법을 사용해야 한다.

동일성 비교 대신에 데이터베이스 동등성 비교를 해보자.

```
member.getId().equals(findMember.getId()) //데이터베이스 식별자 비교
```

데이터베이스 동등성 비교는 엔티티를 영속화해야 식별자를 얻을 수 있다는 문제가 있다. 엔티티를 영속화하기 전에는 식별자 값이 `null`이므로 정확한 비교를 할 수 없다. 물론 식별자 값을 직접 부여하는 방식을 사용할 때는 데이터베이스 식별자 비교도 가능하다. 하지만 항상 식별자를 먼저 부여하는 것을 보장하기는 쉽지 않다.

남은 것은 `equals()`를 사용한 동등성 비교인데, **엔티티를 비교할 때는 비즈니스 키를 활용한 동등성 비교를 권장한다.**

동등성 비교를 위해 `equals()`를 오버라이딩할 때는 비즈니스 키가 되는 필드들을 선택하면 된다. 비즈니스 키가 되는 필드는 보통 중복되지 않고 거의 변하지 않는 데이터베이스 기본 키 후보들이 좋은 대상이다. 만약 주민등록번호가 있다면 중복되지 않고 거의 변경되지 않으므로 좋은 비즈니스 키 대상이다. 이것은 객체 상태에서만 비교하므로 유일성만 보장되면 가끔있는 변경 정도는 허용한다. 따라서 데이터베이스 기본 키 처럼 너무 딱딱하게 정하지 않아도 된다. 예를 들어 회원 엔티티에 이름과 연락처가 같은 회원이 없다면 회원의 이름과 연락처 정도만 조합해서 사용해도 된다.

정리하자면 동일성 비교는 같은 영속성 컨텍스트의 관리를 받는 영속 상태의 엔티티에만 적용할 수 있다. 그렇지 않을 때는 비즈니스 키를 사용한 동등성 비교를 해야 한다.

프록시 심화 주제

프록시는 원본 엔티티를 상속 받아서 만들어지므로 엔티티를 사용하는 클라이언트는 엔티티가 프록시인지 아니면 원본 엔티티인지 구분하지 않고 사용할 수 있다. 따라서 원본 엔티티를 사용하다가 지연 로딩을 하려고 프록시로 변경해도 클라이언트의 비즈니스 로직을 수정하지 않아도 된다.

하지만 프록시를 사용하는 방식의 기술적인 한계로 인해 예상하지 못한 문제들이 발생하기도 하는데, 어떤 문제가 발생하고 어떻게 해결해야 하는지 알아보자.

- 영속성 컨텍스트와 프록시

영속성 컨텍스트는 자신이 관리하는 영속 엔티티의 동일성(identity)을 보장한다. 그럼 프록시로 조회한 엔티티의 동일성도 보장할까? 다음 예제로 확인해보자.

```
@Test
public void 영속성컨텍스트와_프록시() {

    Member newMember = new Member("member1", "회원1");
    em.persist(newMember);
    em.flush();
    em.clear();

    Member refMember = em.getReference(Member.class, "member1");
    Member findMember = em.find(Member.class, "member1");

    System.out.println("refMember  Type = " + refMember.getClass());
    System.out.println("findMember Type = " + findMember.getClass());

    Assert.assertTrue(refMember == findMember);
}
```

출력 결과:

```
refMember  Type = class jpabook.proxy.advanced.Member_$$_jvst843_0
findMember Type = class jpabook.proxy.advanced.Member_$$_jvst843_0
```

먼저 `member1`을 `em.getReference()` 메서드를 사용해서 프록시로 조회했다. 그리고 다음으로 같은 `member1`을 `em.find()`를 사용해서 조회했다. `refMember`는 프록시고 `findMember`는 원본 엔티티이므로 둘은 서로 다른 인스턴스로 생각할 수 있지만 이렇게 되면 영속성 컨텍스트가 영속 엔티티의 동일성을 보장하지 못하는 문제가 발생한다.

그래서 영속성 컨텍스트는 프록시로 조회된 엔티티에 대해서 같은 엔티티를 찾는 요청이 오면 원본 엔티티가 아닌 처음 조회된 프록시를 반환한다. 예제에서 `member1` 엔티티를 프록시로 처음 조회했기 때문에 이후에 `em.find()`를 사용해서 같은 `member1` 엔티티를 찾아도 영속성 컨텍스트는 원본이 아닌 프록시를 반환한다.

`em.find()`로 조회한 `findMember`의 타입을 출력한 결과를 보면 끝에 `$$_jvst843_0`가 붙어있으므로 프록시로 조회된 것을 확인할 수 있다. 그리고 마지막에 `assertTrue` 검증 코드를 통해 `refMember`와 `findMember`가 같은 인스턴스인 것을 알 수 있다.

따라서 프록시로 조회해도 영속성 컨텍스트는 영속 엔티티의 동일성을 보장한다.

원본 먼저 조회하고 나서 프록시로 조회하기

이번에는 반대로 원본 엔티티를 먼저 조회하고 나서 프록시를 조회해보자.

```
@Test
public void 영속성컨텍스트와_프록시2() {

    Member newMember = new Member("member1", "회원1");
    em.persist(newMember);
    em.flush();
    em.clear();

    Member findMember = em.find(Member.class, "member1");
    Member refMember = em.getReference(Member.class, "member1");

    System.out.println("refMember  Type = " + refMember.getClass());
    System.out.println("findMember Type = " + findMember.getClass());

    Assert.assertTrue(refMember == findMember);
}
```

출력 결과:

```
refMember  Type = class jpabook.proxy.advanced.Member
findMember Type = class jpabook.proxy.advanced.Member
```

원본 엔티티를 먼저 조회하면 영속성 컨텍스트는 원본 엔티티를 이미 데이터베이스에서 조회했으므로 프록시를 반환할 이유가 없다. 따라서 `em.getReference()`를 호출해도 프록시가 아닌 원본을 반환한다. 출력 결과를 보면 프록시가 아닌 원본 엔티티가 반환된 것을 확인할 수 있다. 물론 이 경우에도 영속성 컨텍스트는 자신이 관리하는 영속 엔티티의 동일성을 보장한다.

- 프록시 타입 비교

프록시는 원본 엔티티를 상속 받아서 만들어지므로 프록시로 조회한 엔티티의 타입을 비교할 때는 `==` 비교를 하면 안되고 대신에 `instanceof` 를 사용해야 한다.

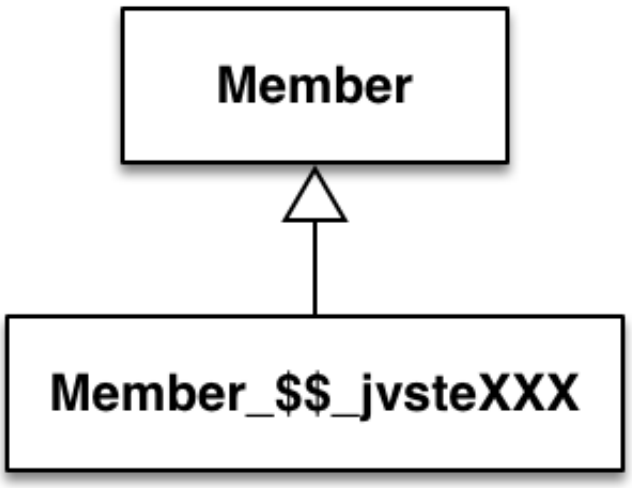

그림 - 프록시 타입 비교

```
@Test
public void 프록시_타입비교() {
    Member newMember = new Member("member1", "회원1");
    em.persist(newMember);
    em.flush();
    em.clear();

    Member refMember = em.getReference(Member.class, "member1");

    System.out.println("refMember Type = " + refMember.getClass());

    Assert.assertFalse(Member.class == refMember.getClass());
    Assert.assertTrue(refMember instanceof Member);
}
```

출력결과:

```
refMember Type = class jpabook.proxy.advanced.Member_$$_jvsteXXX
```

`refMember` 의 타입을 출력해보면 프록시로 조회했으므로 출력결과 끝에 프록시라는 의미의 `_$$_jvsteXXX` 가 붙어있는 것을 확인할 수 있다.

`Member.class == refMember.getClass()` 비교는 부모 클래스와 자식 클래스를 `==` 비교한 것이 된다. 따라서 결과는 `false` 다. 프록시는 원본 엔티티의 자식 타입이므로 `instanceof` 연산을 사용하면 된다.

- 프록시 동등성 비교

엔티티의 동등성을 비교하려면 비즈니스 키를 사용해서 `equals()` 메서드를 오버라이딩하고 비교하면 된다. 그런데 IDE나 외부 라이브러리를 사용해서 구현한 `equals()` 메서드로 엔티티를 비교할 때, 비교 대상이 원본 엔티티면 문제가 없지만 프록시를 비교하면 항상 `false` 를 반환하는 문제가 있다.

다음 예제를 통해서 어떤 문제가 있는지 알아보자.

```
@Entity
public class Member {

    @Id
    private String id;
    private String name;

    ...
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (this.getClass() != obj.getClass()) return false; //(1) //**

        Member member = (Member) obj;

        if (name != null ? !name.equals(member.name) : member.name != null) //(2
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        return name != null ? name.hashCode() : 0;
    }
}
```

회원 엔티티는 `name` 필드를 비즈니스 키로 사용해서 `equals()` 메서드를 오버라이딩 했다.
(`name` 이 중복되는 회원은 없다고 가정한다.)

```
@Test
public void 프록시와_동등성비교() {
    Member saveMember = new Member("member1", "회원1");
    em.persist(saveMember);
    em.flush();
    em.clear();

    Member newMember = new Member("member1", "회원1");
    Member refMember = em.getReference(Member.class, "member1");

    Assert.assertTrue( newMember.equals(refMember) );
}
```

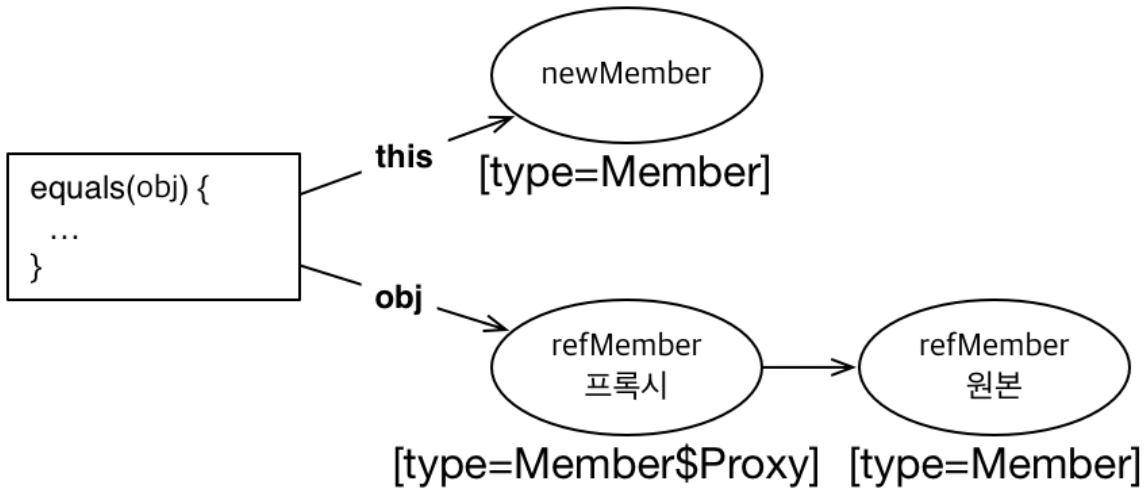

그림 - 프록시 동등성 비교

새로 생성한 회원 `newMember` 와 프록시로 조회한 회원 `refMember` 의 `name` 속성은 둘다 `회원1` 로 같으므로 동등성 비교를 하면 성공할 것 같다. 따라서 `newMember.equals(refMember)` 의 결과는 `true` 를 기대했지만 실행해보면 `false` 가 나오면서 테스트가 실패한다. 참고로 이 테스트를 프록시가 아닌 원본 엔티티를 조회해서 비교하면 성공한다.

프록시와 `equals()` 비교를 할 때는 몇가지 주의점이 있다. 먼저 `equals()` 메서드의 (1) 부분을 보자.

```
if (this.getClass() != obj.getClass()) return false; //(1) //**
```

여기서 `this.getClass() != obj.getClass()`로 타입을 동일성(`==`) 비교한다. 앞서 이야기한데로 프록시는 원본을 상속받은 자식 타입이므로 프록시의 타입을 비교할 때는 `==` 비교가 아닌 `instanceof`를 사용해야 한다. 따라서 다음처럼 변경해야 한다.

```
if (!(obj instanceof Member)) return false;
```

그리고 다음 문제는 `equals()` 메서드의 (2) 부분에 있다.

```
Member member = (Member) obj; //member는 프록시다.

if (name != null ? !name.equals(member.name) : member.name != null) //(2) //**
    return false;
```

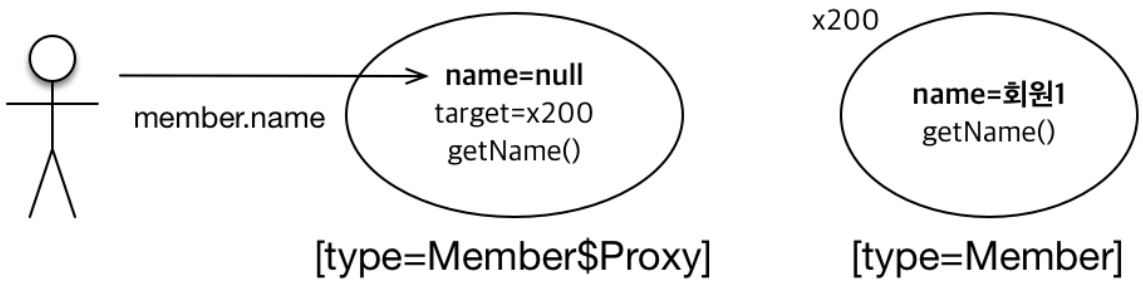

그림 - 프록시 필드 직접 접근

`member.name`을 보면 프록시의 멤버변수에 직접 접근하는데 이 부분을 주의깊게 보아야한다. `equals()` 메서드를 구현할 때는 일반적으로 멤버변수를 직접 비교하는데, 프록시의 경우는 문제가 된다. 프록시는 실제 데이터를 가지고 있지 않다. 따라서 프록시의 멤버변수를 직접 접근하면 아무값도 조회할 수 없으므로 `member.name`의 결과는 `null`이 반환되고 `equals()`는 `false`를 반환한다.

`name` 멤버변수가 `private`이므로 일반적인 상황에서는 프록시의 멤버변수에 직접 접근하는 문제가 발생하지 않지만 `equals()` 메서드는 자신을 비교하기 때문에 `private` 멤버변수에도 접근할 수 있다.

프록시의 데이터를 조회할 때는 접근자(Getter)를 사용해야 한다. 코드를 수정해보자.

```
Member member = (Member) obj;

if (name != null ? !name.equals(member.getName()) : member.getName() != null) //
    return false;
```

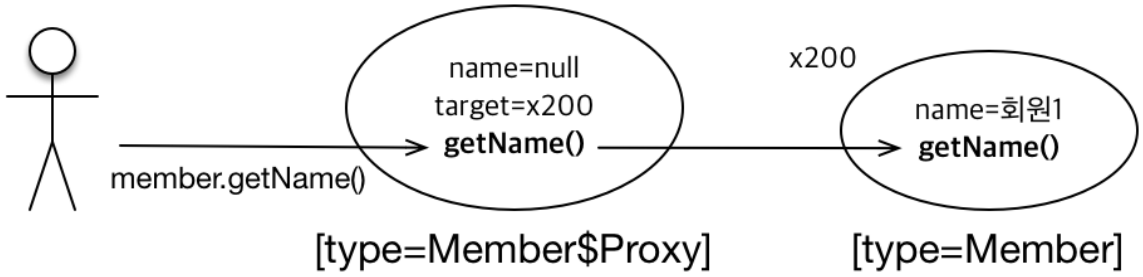

그림 - 프록시 접근자 사용

수정한 전체 코드는 다음과 같다.

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (!(obj instanceof Member)) return false;

    Member member = (Member) obj;

    if (name != null ? !name.equals(member.getName()) : member.getName() != null
        return false;

    return true;
}
```

테스트를 실행하면 `newMember.equals(refMember)`의 결과로 `true`가 반환되고 테스트가 성공한다.

정리하자면 프록시의 동등성을 비교할 때는 다음 사항을 주의해야 한다.

- 프록시의 타입 비교는 `==` 비교 대신에 `instanceof`를 사용해야 한다.
- 프록시의 멤버변수에 직접 접근하면 안되고 대신에 접근자(Getter) 메서드를 사용해야 한다.

- 상속관계와 프록시

상속관계를 프록시로 조회할 때 발생할 수 있는 문제점과 해결방안을 알아보자.

예제에서 사용할 클래스 모델은 다음과 같다.

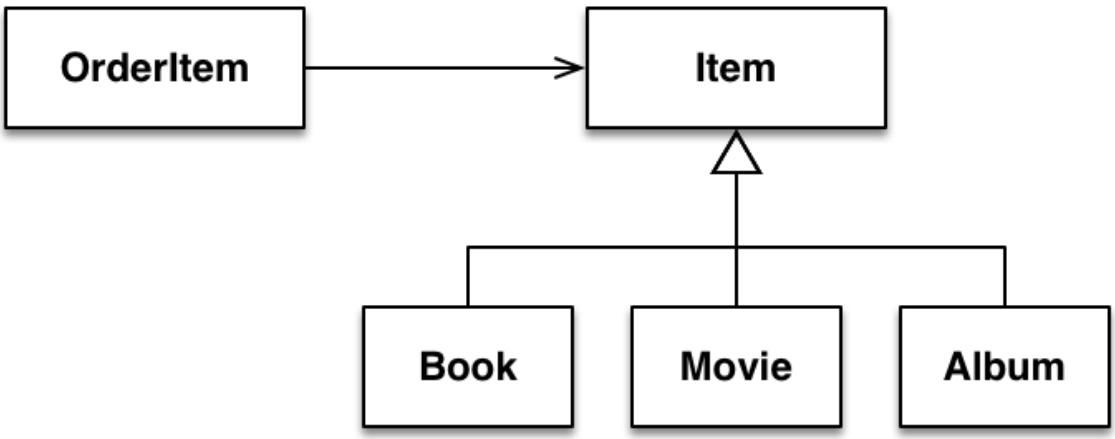

그림 - 상속관계와프록시1

프록시를 부모 타입으로 조회

프록시를 부모 타입으로 조회하면 문제가 발생한다. 예제로 알아보자.

```
@Test
public void 부모타입으로_프록시조회() {

    Book saveBook = new Book();
    saveBook.setName("jpabook");
    saveBook.setAuthor("kim");
    em.persist(saveBook);

    em.flush();
    em.clear();

    Item proxyItem = em.getReference(Item.class, saveBook.getId());
    System.out.println("proxyItem = " + proxyItem.getClass());

    if (proxyItem instanceof Book) {
        System.out.println("proxyItem instanceof Book");
        Book book = (Book) proxyItem;
        System.out.println("책 저자 = " + book.getAuthor());
    }


    Assert.assertFalse( proxyItem.getClass() == Book.class );
    Assert.assertFalse( proxyItem instanceof Book );
    Assert.assertTrue( proxyItem instanceof Item );
}
```

출력결과:

```
proxyItem = class jpabook.proxy.advanced.item.Item_$$_jvstXXX
```

예제는 `Item`을 조회해서 `Book` 타입이면 저자 이름을 출력한다.

코드를 분석해보면 먼저 `Item` 엔티티를 프록시로 찾았다. 그리고 `instanceof` 연산을 사용해서 `proxyItem`이 `Book` 클래스 타입인지 확인한다. `Book` 타입이면 다운캐스팅해서 `Book` 타입으로 변경하고 저자 이름을 출력한다. 그런데 출력결과를 보면 기대와는 다르게 저자가 출력되지 않은 것을 알 수 있다. 다음 그림을 통해 어떤 문제가 있는지 알아보자.

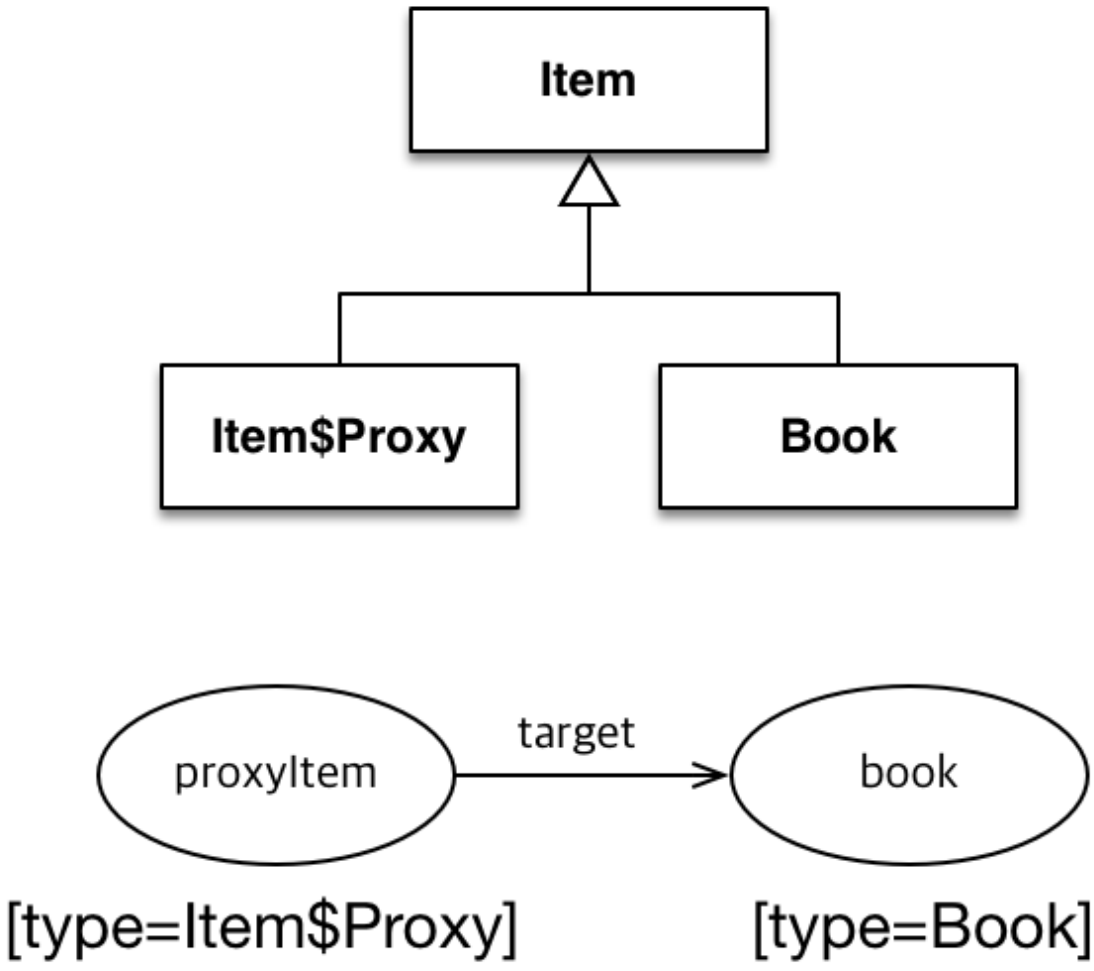

그림 - 상속관계와프록시2

예제에서는 `em.getReference()` 메서드를 사용해서 `Item` 엔티티를 프록시로 조회했다. 이때 실제 조회된 엔티티는 `Book`이므로 `Book` 타입을 기반으로 원본 엔티티 인스턴스가 생성된다. 그런데 `em.getReference()` 메서드에서 `Item` 엔티티를 대상으로 조회했으므로 프록시인 `proxyItem`은 `Item` 타입을 기반으로 만들어진다. 이 프록시 클래스는 원본 엔티티로 `Book` 엔티티를 참조한다.

출력결과와 그림을 보면 `proxyItem`이 `Book`이 아닌 `Item` 클래스를 기반으로 만들어진 것을 확인할 수 있다. 이런 이유로 다음 연산이 기대와 다르게 `false`를 반환한다. 왜냐하면 프록시인 `proxyItem`은 `Item$Proxy` 타입이고 이 타입은 `Book` 타입과 관계가 없기 때문이다.

```
proxyItem instanceof Book //false
```

따라서 직접 다운캐스팅을 해도 문제가 발생한다. 예제 코드에서 if 문을 제거해보자.

```
Book book = (Book) proxyItem; //java.lang.ClassCastException
```

`proxyItem`은 `Book` 타입이 아닌 `Item` 타입을 기반으로한 `Item$Proxy` 타입이다. 따라서 `ClassCastException` 예외가 발생한다.

내용을 정리하면 다음과 같다.

프록시를 부모 타입으로 조회하면 부모의 타입을 기반으로 프록시가 생성되는 문제가 있다.

- `instanceof` 연산을 사용할 수 없다.
- 하위 타입으로 다운캐스팅을 할 수 없다.

프록시를 부모 타입으로 조회하는 문제는 주로 다형성을 다루는 도메인 모델에서 나타난다.

```
@Entity
public class OrderItem {

    @Id @GeneratedValue
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY) //**
    @JoinColumn(name = "ITEM_ID")
    private Item item;

    public Item getItem() {
        return item;
    }
    public void setItem(Item item) {
        this.item = item;
    }

    ...
}
```

`OrderItem`과 `Item`을 지연 로딩으로 설정해서 `item`이 프록시로 조회된다.

```
@Test
public void 상속관계와_프록시_도메인모델() {

    Book book = new Book();
    book.setName("jpabook");
    book.setAuthor("kim");
    em.persist(book);

    OrderItem saveOrderItem = new OrderItem();
    saveOrderItem.setItem(book);
    em.persist(saveOrderItem);

    em.flush();
    em.clear();

    OrderItem orderItem = em.find(OrderItem.class, saveOrderItem.getId());
    Item item = orderItem.getItem();

    System.out.println("item = " + item.getClass());

    Assert.assertFalse( item.getClass() == Book.class );
    Assert.assertFalse( item instanceof Book );
    Assert.assertTrue(  item instanceof Item );
}
```

출력결과:

```
item = class jpabook.proxy.advanced.item.Item_$$_jvstffa_3
```

`OrderItem`과 연관된 `Item`을 지연 로딩으로 설정했으므로 출력결과를 보면 `item`이 프록시로 조회된 것을 확인할 수 있다. 따라서 `item instanceof Book` 연산도 `false`를 반환한다.

그렇다면 상속관계에서 발생하는 프록시 문제를 어떻게 해결해야 할까? 문제 해결 방법을 하나씩 알아보자.

– JPQL로 대상 직접 조회하기

가장 간단한 방법은 처음부터 자식 타입을 직접 조회해서 필요한 연산을 하면 된다.

```
Book jpqlBook = em.createQuery("select b from Book b where b.id=:bookId", Book.c
        .setParameter("bookId", item.getId())
        .getSingleResult();
```

– 프록시 벗기기

하이버네이트가 제공하는 기능을 사용하면 프록시에서 원본 엔티티를 가져올 수 있다.

```
    ...
    Item item = orderItem.getItem();
    Item unProxyItem = unProxy(item);

    if (unProxyItem instanceof Book) {
        System.out.println("proxyItem instanceof Book");
        Book book = (Book) unProxyItem;
        System.out.println("책 저자 = " + book.getAuthor());
    }

    Assert.assertTrue(item != unProxyItem);
}

//하이버네이트가 제공하는 프록시에서 원본 엔티티를 찾는 기능을 사용하는 메서드
public static <T> T unProxy(Object entity) {
    if (entity instanceof HibernateProxy) {
        entity = ((HibernateProxy) entity)
                        .getHibernateLazyInitializer()
                        .getImplementation();
    }
    return (T) entity;
}
```

출력결과:

```
proxyItem instanceof Book
책 저자 = shj
```

이 장 처음에 설명했듯이 영속성 컨텍스트는 한번 프록시로 노출한 엔티티는 계속 프록시로 노출한다. 그래야 영속성 컨텍스트가 영속 엔티티의 동일성을 보장할 수 있고, 클라이언트는 조회한 엔티티가 프록시인지 아닌지 구분하지 않고 사용할 수 있다. 그런데 이 방법은 프록시에서 원본 엔티티를 직접 꺼내기 때문에 프록시와 원본 엔티티의 동일성 비교가 실패한다는 문제점이 있다. 따라서 다음 연산의 결과는 `false` 다.

```
item == unProxyItem //false
```

이 방법을 사용할 때는 원본 엔티티가 꼭 필요한 곳에서 잠깐 사용하고 다른 곳에서 사용되지 않도록 하는 것이 중요하다. 참고로 원본 엔티티의 값을 직접 변경해도 변경 감지 기능은 동작한다.

– 기능을 위한 별도의 인터페이스 제공

이번에는 특정 기능을 제공하는 인터페이스를 사용하도록 해보자.

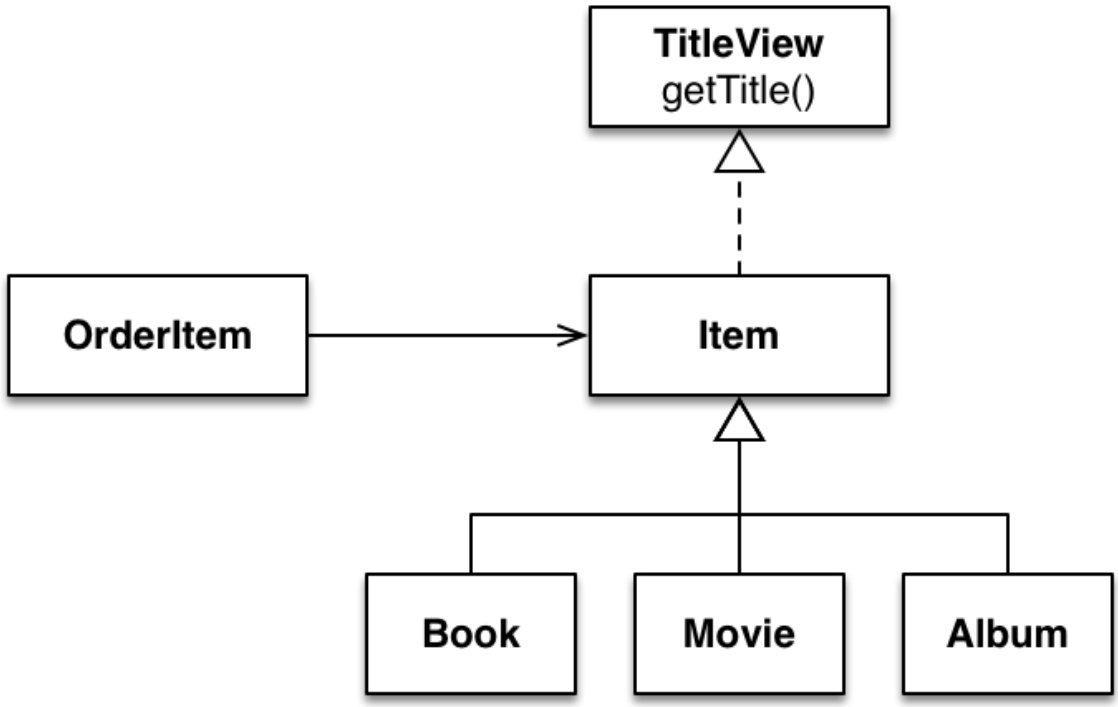

그림 - 상속관계와프록시3

```
public interface TitleView {
    String getTitle();
}

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "DTYPE")
public abstract class Item implements TitleView {

    @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id;

    private String name;
```

```
    private int price;
    private int stockQuantity;

    //Getter, Setter
}

@Entity
@DiscriminatorValue("B")
public class Book extends Item {

    private String author;
    private String isbn;

    @Override
    public String getTitle() {
        return "[제목:" + getName() + " 저자:" + author + "]";
    }
}

@Entity
@DiscriminatorValue("M")
public class Movie extends Item {

    private String director;
    private String actor;

    //Getter, Setter

    @Override
    public String getTitle() {
        return "[제목:" + getName() + " 감독:" + director + " 배우:" + actor + "]";
    }
}
```

`TitleView`라는 공통 인터페이스를 만들고 인터페이스의 `getTitle()` 메서드를 각각 구현했다.

====== 사용 클래스 ======

```
@Entity
public class OrderItem {

    @Id @GeneratedValue
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "ITEM_ID")
    private Item item;
```

```
    ...

    public void printItem() {   //**
        System.out.println("TITLE="+item.getTitle());
    }
}
```

====== 사용 코드 ======

```
OrderItem orderItem = em.find(OrderItem.class, saveOrderItem.getId());
orderItem.printItem();
```

`Item`의 구현체에 따라 각각 다른 `getTitle()` 메서드가 호출된다. 예를 들어 `Book`을 조회했으면 다음 결과가 출력된다.

```
TITLE=[제목:jpabook 저자:kim]
```

이처럼 인터페이스를 제공하고 각각의 클래스가 자신에 맞는 기능을 구현하는 것은 다형성을 활용하는 좋은 방법이다. 이후 다양한 상품 타입이 추가되어도 `Item`을 사용하는 `OrderItem`의 코드는 수정하지 않아도 된다. 그리고 이 방법은 클라이언트 입장에서 대상 객체가 프록시인지 아닌지를 고민하지 않아도 되는 장점이 있다.

이 방법을 사용할 때는 프록시의 특징 때문에 프록시의 대상이 되는 타입에 인터페이스를 적용해야 한다. 여기서는 `Item`이 프록시의 대상이므로 `Item`이 인터페이스를 받아야한다.

– 비지터 패턴 사용

이번에는 비지터 패턴(Visitor Pattern)[1]을 사용해서 상속관계와 프록시 문제를 해결해보자.

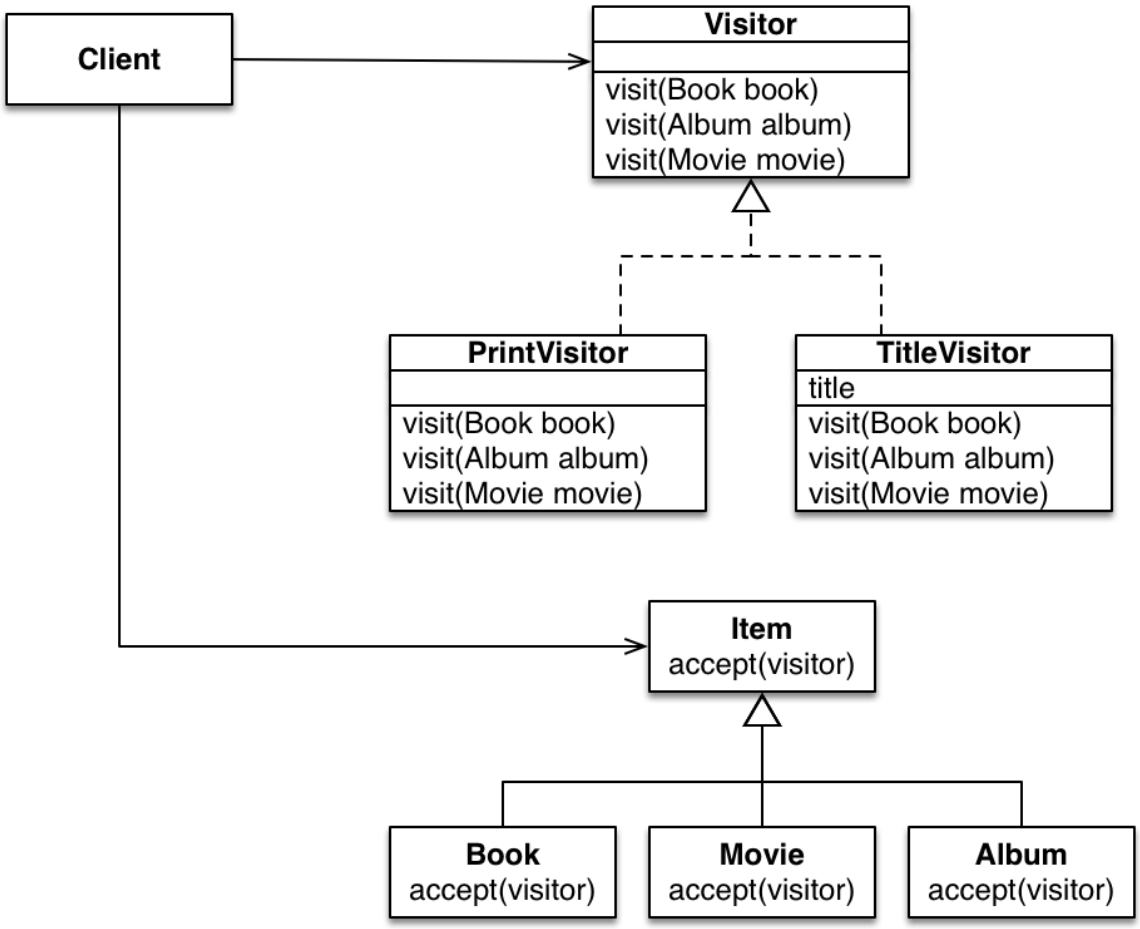

그림 - 상속관계와프록시4

비지터 패턴은 `Visitor`와 `Visitor`를 받아들이는 대상 클래스로 구성되는데 여기서는 `Item`이 `accept(visitor)` 메서드를 사용해서 `Visitor`를 받아들인다. 그리고 `Item`은 단순히 `Visitor`를 받아들이기만 하고 실제 로직은 `Visitor`가 처리하게 된다.

Visitor 정의와 구현

====== Visitor ======

```
public interface Visitor {

    void visit(Book book);
    void visit(Album album);
    void visit(Movie movie);
}
```

`Visitor`에는 `visit()`라는 메서드를 정의하고 모든 대상 클래스를 받아들이도록 작성하면 된다. 여기서는 `Book`, `Album`, `Movie`를 대상 클래스로 사용한다.

```
public class PrintVisitor implements Visitor {

    @Override
    public void visit(Book book) {
        //넘어오는 book은 Proxy가 아닌 원본 엔티티다.
        System.out.println("book.class = " + book.getClass());
        System.out.println("[PrintVisitor] [제목:" + book.getName() + " 저자:" + b
    }

    @Override
    public void visit(Album album) {...}
    @Override
    public void visit(Movie movie) {...}
}

public class TitleVisitor implements Visitor {

    private String title;

    public String getTitle() {
        return title;
    }

    @Override
    public void visit(Book book) {
        title = "[제목:" + book.getName() + " 저자:" + book.getAuthor() + "]";
    }

    @Override
    public void visit(Album album) {...}
    @Override
    public void visit(Movie movie) {...}
}
```

`Visitor`의 구현 클래스로 대상 클래스의 내용을 출력해주는 `PrintVisitor`와 대상 클래스의 제목을 보관하는 `TitleVisitor`를 작성했다.

대상 클래스 작성

`Item`에 `Visitor`를 받아들일 수 있도록 `accept(visitor)` 메서드를 추가했다.

```
public abstract void accept(Visitor visitor);
```

```
@Entity
```

```
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "DTYPE")
public abstract class Item {

    @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id;

    private String name;

    ...

    public abstract void accept(Visitor visitor); //**
}

@Entity
@DiscriminatorValue("B")
public class Book extends Item {

    private String author;
    private String isbn;
    //Getter, Setter
    public String getAuthor() {
        return author;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

@Entity
@DiscriminatorValue("M")
public class Movie extends Item {
    ...
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

@Entity
@DiscriminatorValue("A")
public class Album extends Item {
    ...
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

각각의 자식 클래스들은 부모에 정의한 `accept(visitor)` 메서드를 구현했는데, 구현 내용은 단순히 파리미터로 넘어온 `Visitor`의 `visit(this)` 메서드를 호출하면서 자신(`this`)을 파라미터로 넘기는 것이 전부다. 이렇게 해서 실제 로직 처리를 `visitor`에 위임한다.

실제 어떻게 사용하는지 알아보자.

====== 사용 코드 ======

```
@Test
public void 상속관계와_프록시_VisitorPattern() {
    ...
    OrderItem orderItem = em.find(OrderItem.class, orderItemId);
    Item item = orderItem.getItem();

    //PrintVisitor
    item.accept(new PrintVisitor());
}
```

출력결과:

```
book.class = class jpabook.proxy.advanced.item.Book
[PrintVisitor] [제목:jpabook 저자:kim]
```

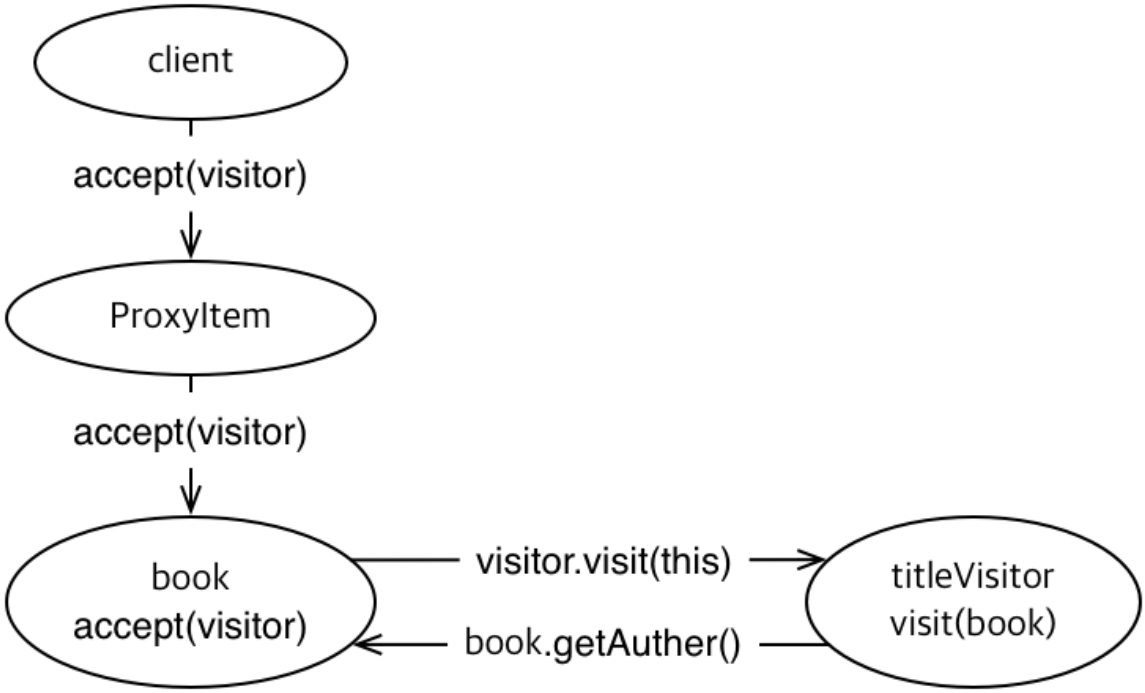

그림 - 상속관계와프록시5

사용하는 코드를 분석해보자. 먼저 `item.accept()` 메서드를 호출하면서 파라미터로 `PrintVisitor`를 넘겨주었다. `item`은 프록시이므로 먼저 프록시(`ProxyItem`)가 `accept()` 메서드를 받고 원본 엔티티(`book`)의 `accept()`를 실행한다. 원본 엔티티는 다음 코드를 실행해서 자신(`this`)을 `visitor`에 파리미터로 넘겨준다.

```
public void accept(Visitor visitor) {
    visitor.visit(this);
}
```

`visitor`가 `PrintVisitor` 타입이므로 `PrintVisitor.visit(this)` 메서드가 실행되는데 이때 `this`가 `Book` 타입이므로 `visit(Book book)` 메서드가 실행된다.

```
public class PrintVisitor implements Visitor {

    public void visit(Book book) {
        //넘어오는 book은 Proxy가 아닌 원본 엔티티다.
        System.out.println("book.class = " + book.getClass());
        System.out.println("[PrintVisitor] [제목:" + book.getName() + " 저자:" + b
    }
    public void visit(Album album) {...}
    public void visit(Movie movie) {...}
}
```

다음 출력 결과를 보면 `Visitor.visit()`에서 파라미터로 넘어오는 엔티티는 프록시가 아니라 실제 원본 엔티티인 것을 확인할 수 있다.

```
book.class = class jpabook.proxy.advanced.item.Book
```

이렇게 비지터 패턴을 사용하면 프록시에 대한 걱정 없이 안전하게 원본 엔티티에 접근할 수 있고 `instanceof`나 타입캐스팅 없이 코드를 구현할 수 있는 장점이 있다.

비지터 패턴과 확장성

`TitleVisitor`를 사용하도록 변경해보자.

```
//TitleVisitor
TitleVisitor titleVisitor = new TitleVisitor();
item.accept(titleVisitor);

String title = titleVisitor.getTitle();
System.out.println("TITLE=" + title);
```

출력결과:

```
book.class = class jpabook.proxy.advanced.item.Book
TITLE=[제목:jpabook 저자:kim]
```

비지터 패턴은 새로운 기능이 필요할 때 `Visitor`만 추가하면 된다. 따라서 기존 코드의 구조를 변경하지 않고 기능을 추가할 수 있는 장점이 있다.

비지터 패턴 정리

장점

- 프록시에 대한 걱정 없이 안전하게 원본 엔티티에 접근할 수 있다.
- `instanceof`와 타입캐스팅 없이 코드를 구현할 수 있다.
- 알고리즘과 객체 구조를 분리해서 구조를 수정하지 않고 새로운 동작을 추가할 수 있다.

단점

- 너무 복잡하고 더블 디스패치[2]를 사용하기 때문에 이해하기 어렵다.

- 객체 구조가 변경 되면 모든 `Visitor`를 수정해야 한다.

지금까지 상속관계에서 발생하는 프록시의 문제점과 다양한 해결방법을 알아보았다. 이 방법들을 조금씩 응용하면 프록시로 인해 발생하는 문제는 어렵지 않게 해결할 수 있을 것이다.

---

1. http://en.wikipedia.org/wiki/Visitor_pattern ↩
2. http://en.wikipedia.org/wiki/Double_dispatch ↩