객체 지향 쿼리 심화

이 장에서는 객체 지향 쿼리와 관련된 다양한 고급 주제를 알아보자.

벌크 연산

엔티티를 수정하려면 영속성 컨텍스트의 변경 감지 기능이나 병합을 사용하고, 삭제하려면 `EntityManager.remove()` 메서드를 사용한다. 하지만 이 방법으로 수백 개 이상의 엔티티를 하나씩 처리하기에는 시간이 너무 오래 걸린다. 이럴 때 여러 건을 한 번에 수정하거나 삭제하는 벌크 연산을 사용하면 된다.

예를 들어 재고가 10개 미만인 모든 상품의 가격을 10% 상승시키려면 다음처럼 벌크연산을 사용하면 된다.

```
String qlString =
        "update Product p " +
        "set p.price = p.price * 1.1 " +
        "where p.stockAmount < :stockAmount";
```

```
int resultCount = em.createQuery(qlString)
                    .setParameter("stockAmount", 10)
                    .executeUpdate(); //**
```

연산은 `executeUpdate()` 메서드를 사용한다. 이 메서드는 벌크 연산으로 영향을 받은 엔티티 건수를 반환한다.

삭제도 같은 메서드를 사용한다. 다음은 가격이 100원 미만인 상품을 삭제하는 코드다.

```
String qlString =
        "delete from Product p " +
        "where p.price < :price";

int resultCount = em.createQuery(qlString)
                    .setParameter("price", 100)
                    .executeUpdate();
```

> **참고:** JPA 표준은 아니지만 하이버네이트는 INSERT 벌크 연산도 지원한다. 다음 코드는 100원 미만의 모든 상품을 선택해서 `ProductTemp`에 저장한다.
>
> ```
> String qlString =
>         "insert into ProductTemp(id, name, price, stockAmount) " +
>         "select p.id, p.name, p.price, p.stockAmount from Product p "
>         "where p.price < :price";
>
> int resultCount = em.createQuery(qlString)
>                     .setParameter("price", 100)
>                     .executeUpdate();
> ```

- 벌크 연산의 주의점

벌크 연산을 사용할 때는 **벌크 연산이 영속성 컨텍스트를 무시하고 데이터베이스에 직접 쿼리**한다는 점에 주의해야 한다.

벌크 연산시 어떤 문제가 발생할 수 있는지 다음 예제를 통해 알아보자.

데이터베이스에는 가격이 1000원인 상품A(productA)가 있다.

```
//1. 상품A 조회(상품A의 가격은 1000원이다.)
Product productA = em.createQuery("select p from Product p where p.name = :name'
        .setParameter("name", "productA")
        .getSingleResult();

//출력결과 : 1000
System.out.println("productA 수정 전 = " + productA.getPrice());

//2. 벌크 연산 수행으로 모든 상품 가격 10% 상승
em.createQuery("update Product p set p.price = p.price * 1.1")
    .executeUpdate();

//3. 출력결과 : 1000
System.out.println("productA 수정 후 = " + productA.getPrice());
```

1. 가격이 1000원인 상품A를 조회했다. 조회된 상품A는 영속성 컨텍스트에서 관리된다.
2. 벌크 연산으로 모든 상품의 가격을 10% 상승시켰다. 따라서 상품A의 가격은 1100원이 되어야 한다.
3. 벌크 연산을 수행한 후에 상품A 의 가격을 출력하면 기대했던 1100원이 아니라 1000원이 출력된다.

이 상황을 그림으로 분석해보자.

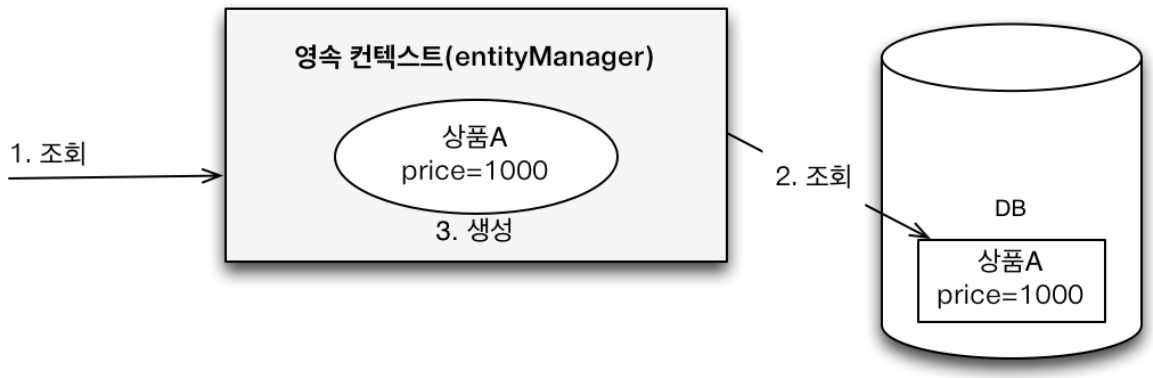

그림 9 I 벌크연산 직전

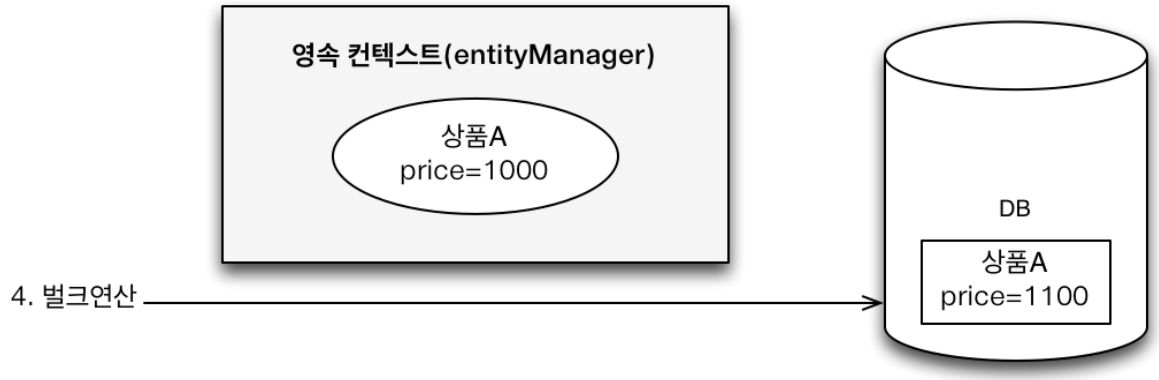

그림 9 I 벌크연산 수행 후

벌크 연산 수행 후 그림을 보자. 벌크 연산은 영속성 컨텍스트를 통하지 않고 데이터베이스에 직접 쿼리한다. 따라서 영속성 컨텍스트에 있는 상품A와 데이터베이스에 있는 상품A의 가격이 다를 수 있다. 따라서 벌크 연산은 주의해서 사용해야 한다.

이런 문제를 해결하는 다양한 방법을 알아보자.

`em.refresh()` 사용

벌크 연산을 수행한 직후에 정확한 상품A 엔티티를 사용해야 한다면 `em.refresh()`를 사용해서 데이터베이스에서 상품A를 다시 조회하면 된다.

```
em.refresh(productA); // 데이터베이스에서 상품A를 다시 조회한다.
```

벌크 연산 먼저 실행

가장 실용적인 해결책은 벌크 연산을 가장 먼저 실행하는 것이다. 예를 들어 위에서 벌크 연산을 먼저 실행하고 나서 상품A를 조회하면 벌크 연산으로 이미 변경된 상품A를 조회하게 된다. 이 방법은 JPA와 JDBC를 함께 사용할 때도 유용하다.

벌크 연산 수행 후 영속성 컨텍스트 초기화

벌크 연산을 수행한 직후에 바로 영속성 컨텍스트를 초기화해서 영속성 컨텍스트에 남아있는 엔티티를 제거하는 것도 좋은 방법이다. 그렇지 않으면 엔티티를 조회할 때 영속성 컨텍스트에 남아 있는 엔티티를 조회할 수 있는데 이 엔티티에는 벌크 연산이 적용되어 있지 않다. 영속성 컨텍스트를 초기화하면 이후 엔티티를 조회할 때 벌크 연산이 적용된 데이터베이스에서 엔티티를 조회한다.

벌크 연산 정리

벌크 연산은 영속성 컨텍스트와 2차 캐시를 무시하고 데이터베이스에 직접 실행한다. 따라서 영속성 컨텍스트와 데이터베이스 간에 데이터 차이가 발생할 수 있으므로 주의해서 사용해야 한다. 가능하면 벌크 연산을 가장 먼저 수행하는 것이 좋고 상황에 따라 영속성 컨텍스트를 초기화하는 것도 필요하다.

영속성 컨텍스트와 JPQL

- 쿼리 후 영속 상태인 것과 아닌 것

JPQL의 조회 대상은 엔티티, 임베디드 타입, 값 타입 같이 다양한 종류가 있다. JPQL로 **엔티티를 조회하면 영속성 컨텍스트에서 관리되지만 엔티티가 아니면 영속성 컨텍스트에서 관리되지 않는다.**

```
select m from Member m //엔티티 조회 (관리O)
select o.address from Order o //임베디드 타입 조회 (관리X)
select m.id, m.username from Member m //단순 필드 조회 (관리X)
```

예를 들어 임베디드 타입은 조회해서 값을 변경해도 영속성 컨텍스트가 관리하지 않으므로 변경 감지에 의한 수정이 발생하지 않는다. 물론 엔티티를 조회하면 해당 엔티티가 가지고 있는 임베디드 타입은 함께 수정된다.

정리하자면 **JPQL로 조회한 엔티티만 영속성 컨텍스트가 관리한다.**

- JPQL로 조회한 엔티티와 영속성 컨텍스트

그런데 만약 다음 예제처럼 영속성 컨텍스트에 회원1이 이미 있는데, JPQL로 회원1을 다시 조회하면 어떻게 될까?

```
em.find(Member.class, "member1"); //회원1 조회

//엔티티 쿼리 조회 결과가 회원1, 회원2
List<Member> resultList = em.createQuery("select m from Member m", Member.class)
                            .getResultList();
```

JPQL로 데이터베이스에서 조회한 엔티티가 영속성 컨텍스트에 이미 있으면 JPQL로 조회한 결과를 버리고 대신에 영속성 컨텍스트에 있던 엔티티를 반환한다. 이때 식별자 값을 사용해서 비교한다.

이해를 돕기 위해 그림으로 설명하겠다.

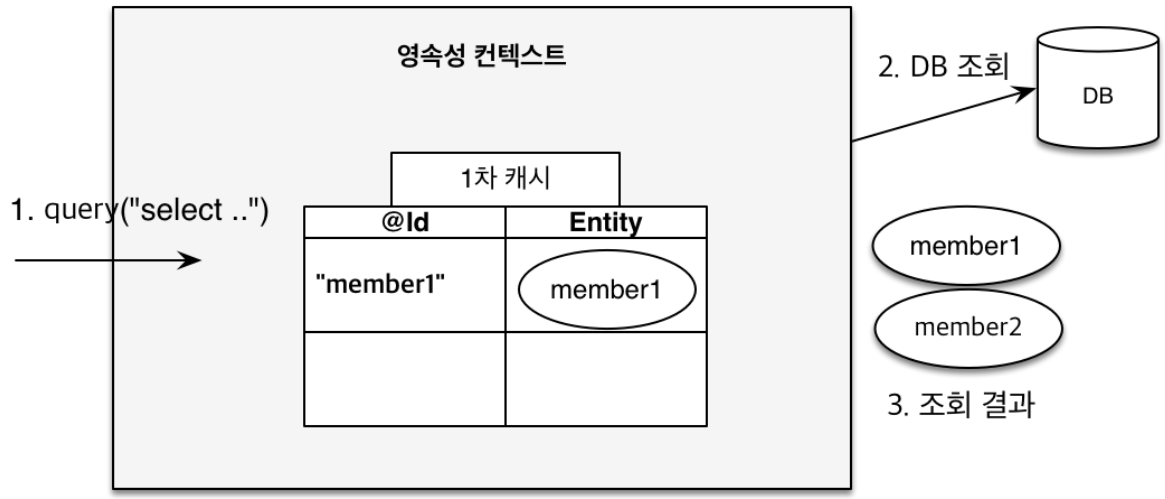

그림 엔티티쿼리1 I 엔티티 쿼리와 결과

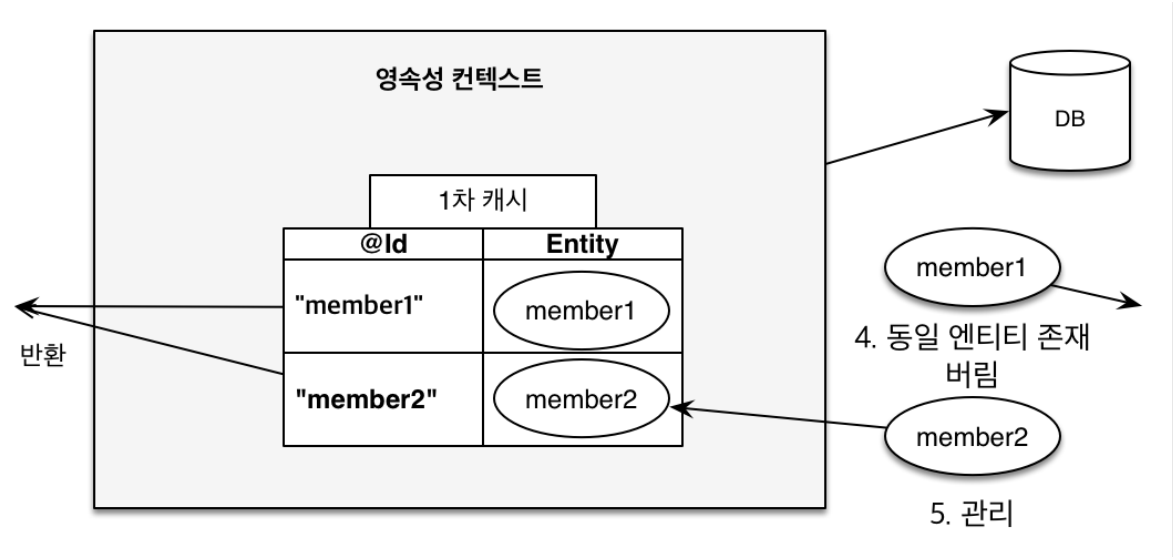

그림 엔티티쿼리2 I 결과 비교와 리턴

1. JPQL을 사용해서 조회를 요청한다.
2. JPQL은 SQL로 변환되어 데이터베이스를 조회한다.
3. 조회한 결과와 영속성 컨텍스트를 비교한다.
4. 식별자 값을 기준으로 `member1` 은 이미 영속성 컨텍스트에 있으므로 버리고 기존에 있던 `member1` 이 반환 대상이 된다.
5. 식별자 값을 기준으로 `member2` 는 영속성 컨텍스트에 없으므로 영속성 컨텍스트에 추가한다.
6. 쿼리 결과인 `member1` , `member2` 를 반환한다. 여기서 `member1` 은 쿼리 결과가 아닌 영속성 컨텍스트에 있던 엔티티다.

그림을 통해 다음 2가지를 확인할 수 있다.

- JPQL로 조회한 엔티티는 영속 상태다.
- 영속성 컨텍스트에 이미 존재하는 엔티티가 있으면 기존 엔티티를 반환한다.

그런데 왜 데이터베이스에서 새로 조회한 `member1`을 버리고 영속성 컨텍스트에 있는 기존 엔티티를 반환하는 것일까? JPQL로 조회한 새로운 엔티티를 영속성 컨텍스트에 하나 더 추가하거나 기존 엔티티를 새로 검색한 엔티티로 대체하면 어떤 문제가 있는 것일까?

1. 새로운 엔티티를 영속성 컨텍스트에 하나 더 추가한다.
2. 기존 엔티티를 새로 검색한 엔티티로 대체한다.
3. 기존 엔티티는 그대로 두고 새로 검색한 엔티티를 버린다.

영속성 컨텍스트는 기본 키 값을 기준으로 엔티티를 관리한다. 따라서 같은 기본 키 값을 가진 엔티티는 등록할 수 없으므로 1번은 아니다. 2번은 언뜻 보면 합리적인 것 같지만, 영속성 컨텍스트에 수정 중인 데이터가 사라질 수 있으므로 위험하다. 영속성 컨텍스트는 엔티티의 동일성을 보장한다. 따라서 영속성 컨텍스트는 3번으로 동작한다.

영속성 컨텍스트는 영속 상태인 엔티티의 동일성을 보장한다. `em.find()`로 조회하든 JPQL을 사용하든 영속성 컨텍스트가 같으면 동일한 엔티티를 반환한다.

- find() vs JPQL

`em.find()` 메서드는 엔티티를 영속성 컨텍스트에서 먼저 찾고 없으면 데이터베이스에서 찾는다. 따라서 해당 엔티티가 영속성 컨텍스트에 있으면 메모리에서 바로 찾으므로 성능상 이점이 있다. (그래서 1차 캐시라 부른다.)

```
//최초 조회, 데이터베이스에서 조회
Member member1 = em.find(Member.class, 1L);
//두 번째 조회, 영속성 컨텍스트에 있으므로 데이터베이스를 조회하지 않음
Member member2 = em.find(Member.class, 1L);

//member1 == member2는 주소 값이 같은 인스턴스
```

그렇다면 JPQL은 어떤 방식으로 동작할까? 다음 코드를 보자.

```
//첫 번째 호출 : 데이터베이스에서 조회
Member member1 = em.createQuery("select m from Member m where m.id = :id", Membe
        .setParameter("id", 1L)
        .getSingleResult();

//두 번째 호출 : 데이터베이스에서 조회
Member member2 = em.createQuery("select m from Member m where m.id = :id", Membe
        .setParameter("id", 1L)
        .getSingleResult();
```

```
  //member1 == member2는 주소값이 같은 인스턴스
```

`em.find()`를 2번 사용한 로직과 마찬가지로 주소 값이 같은 인스턴스를 반환하고 결과도 같다. 하지만 내부 동작방식은 조금 다르다.

JPQL은 항상 데이터베이스에 SQL을 실행해서 결과를 조회한다.

`em.find()` 메서드는 영속성 컨텍스트에서 엔티티를 먼저 찾고 없으면 데이터베이스를 조회하지만 JPQL을 사용하면 데이터베이스를 먼저 조회한다. (JPA 구현체 개발자 입장에서 `em.find()` 메서드는 파라미터로 식별자 값을 넘기기 때문에 영속성 컨텍스트를 조회하기 쉽지만, JPQL을 분석해서 영속성 컨텍스트를 조회하는 것은 쉬운일이 이니다. 따라서 JPQL로 쿼리한 결과 값을 사용한다.)

이 코드에서 첫 번째 JPQL을 호출하면 데이터베이스에서 회원 엔티티(`id=1L`)를 조회하고 영속성 컨텍스트에 등록한다. 두 번째 JPQL을 호출하면 데이터베이스에서 같은 회원 엔티티(`id=1L`)를 조회한다. 이때 영속성 컨텍스트에 이미 조회한 같은 엔티티가 있다. 앞서 이야기 한데로 새로 검색한 엔티티는 버리고 영속성 컨텍스트에 있는 기존 엔티티를 반환한다.

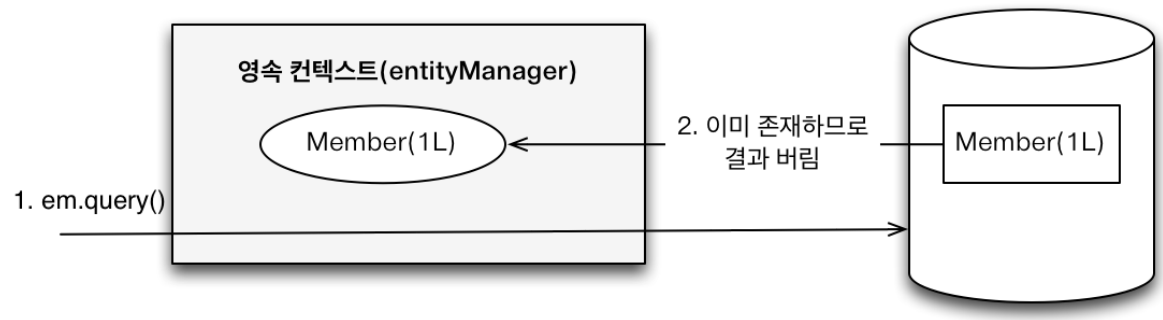

그림 9 I 쿼리 결과가 영속성 컨텍스트에 존재

JPQL의 특징 정리

- JPQL은 항상 데이터베이스를 조회한다.
- JPQL로 조회한 엔티티는 영속 상태다.
- 영속성 컨텍스트에 이미 존재하는 엔티티가 있으면 기존 엔티티를 반환한다.

JPQL과 플러시 모드

시작하기 전에 플러시 모드를 복습해보자. 플러시는 영속성 컨텍스트의 변경 내역을 데이터베이스에 동기화하는 것이다. JPA는 플러시가 일어날 때 영속성 컨텍스트에 등록, 수정, 삭제한 엔티티를 찾아서 INSERT, UPDATE, DELETE SQL을 만들어 데이터베이스에 반영한다. 플러시를 호출하려면

`em.flush()` 메서드를 직접 사용해도 되지만 보통 플러시 모드(FlushMode)에 따라 커밋하기 직전이나 쿼리 실행 직전에 자동으로 플러시가 호출된다.

```
em.setFlushMode(FlushModeType.AUTO);  //커밋 또는 쿼리 실행시 플러시 (기본값)
em.setFlushMode(FlushModeType.COMMIT);//커밋시에만 플러시
```

플러시 모드는 `FlushModeType.AUTO`가 기본값이다. 따라서 JPA는 트랜잭션 커밋 직전이나 쿼리 실행 직전에 자동으로 플러시를 호출한다. 다른 옵션으로는 `FlushModeType.COMMIT`이 있는데 이 모드는 커밋시에만 플러시를 호출하고 쿼리 실행시에는 플러시를 호출하지 않는다. 이 옵션은 성능 최적화를 위해 꼭 필요할 때만 사용해야 한다.

- 쿼리와 플러시 모드

JPQL은 영속성 컨텍스트에 있는 데이터를 고려하지 않고 데이터베이스에서 데이터를 조회한다. 따라서 JPQL을 실행하기 전에 영속성 컨텍스트의 내용을 데이터베이스에 반영해야 한다. 그렇지 않으면 의도하지 않은 결과가 발생할 수 있다. 다음 그림과 예제를 보자.

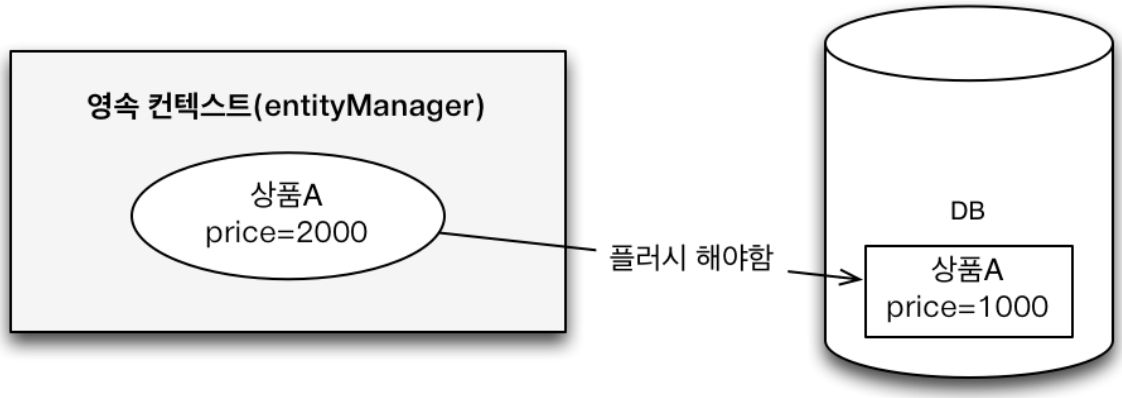

그림 9 I 영속성 컨텍스트가 아직 플러시 되지 않은 상황

```
//가격을 1000->2000원으로 변경
product.setPrice(2000);

//가격이 2000원인 상품 조회
Product product2 = em.createQuery("select p from Product p where p.price = 2000'
        .getSingleResult();
```

`product.setPrice(2000)`를 호출하면 영속성 컨텍스트의 상품 엔티티는 가격이 1000원에서 2000원으로 변경되지만 데이터베이스에는 1000원인 상태로 남아있다. 다음으로 JPQL을 호출해서 가격이 2000원인 상품을 조회했는데 이때 플러시 모드를 따로 설정하지 않으면 플러시 모드가 `AUTO`이므로 쿼리 실행 직전에 영속성 컨텍스트가 플러시 된다. 따라서 방금 2000원으로 수정한 상품을 조회할 수 있다.

만약 이런 상황에서 플러시 모드를 `COMMIT`으로 설정하면 쿼리시에는 플러시 하지 않으므로 방금 수정한 데이터를 조회할 수 없다. 이때는 직접 `em.flush()`를 호출하거나 다음 코드처럼 `Query` 객체에 플러시 모드를 설정해 주면 된다.

```
em.setFlushMode(FlushModeType.COMMIT); //커밋시에만 플러시

//가격을 1000->2000원으로 변경
product.setPrice(2000);

//1. em.flush() 직접 호출

//가격이 2000원인 상품 조회
Product product2 = em.createQuery("select p from Product p where p.price = 2000"
        .setFlushMode(FlushModeType.AUTO) //2. setFlushMode() 설정
        .getSingleResult();
```

코드를 보면 첫 줄에서 플러시 모드를 `COMMIT`으로 설정했다. 따라서 쿼리를 실행할 때 플러시를 자동으로 호출하지 않는다. 만약 쿼리 실행전에 플러시를 호출하고 싶으면 주석으로 처리해둔 1. `em.flush()`의 주석을 풀어서 수동으로 플러시 하거나 아니면 2. `setFlushMode()`로 해당 쿼리에서만 사용할 플러시 모드를 `AUTO`로 변경하면 된다. 이렇게 쿼리에 설정하는 플러시 모드는 엔티티 매니저에 설정하는 플러시 모드보다 우선권을 가진다.

플러시 모드의 기본값은 `AUTO`로 설정되어 있으므로 일반적인 상황에서는 방금 설명한 내용을 고민하지 않아도 된다. 그럼 왜 `COMMIT` 모드를 사용하는 것일까?

- 플러시 모드와 최적화

```
em.setFlushMode(FlushModeType.COMMIT)
```

`FlushModeType.COMMIT` 모드는 트랜잭션을 커밋할 때만 플러시하고 쿼리를 실행할 때는 플러시하지 않는다. 따라서 JPA 쿼리를 사용할 때 영속성 컨텍스트에는 있지만 아직 데이터베이스에 반영하지 않은 데이터를 조회할 수 없다. 이런 상황은 잘못하면 데이터 무결성에 심각한 피해를 줄 수 있다. 그럼에도

다음과 같이 플러시가 너무 자주 일어나는 상황에 이 모드를 사용하면 쿼리시 발생하는 플러시 횟수를 줄여서 성능을 최적화할 수 있다.

```
//비즈니스 로직
등록()
쿼리() //플러시
등록()
쿼리() //플러시
등록()
쿼리() //플러시
커밋() //플러시
```

- `FlushModeType.AUTO` : 쿼리와 커밋할 때 총 4번 플러시한다.
- `FlushModeType.COMMIT` : 커밋시에만 1번 플러시한다.

JPA를 사용하지 않고 JDBC를 직접 사용해서 SQL을 실행할 때도 플러시 모드를 고민해야 한다. JPA를 통하지 않고 JDBC로 쿼리를 직접 실행하면 JPA는 JDBC가 실행한 쿼리를 인식할 방법이 없다. 따라서 별도의 JDBC 호출은 플러시 모드를 AUTO 설정해도 플러시가 일어나지 않는다. 이때는 JDBC로 쿼리를 실행하기 직전에 `em.flush()`를 호출해서 영속성 컨텍스트의 내용을 데이터베이스에 동기화하는 것이 안전하다.