3장 영속성 관리

기간 : 21.12.09 ~ 21.12.16

페이지 : 90p ~ 119p

▼ 목차

- 3.1 엔티티 매니저 팩토리와 앤티티 매니저
  - **엔티티 매니저 팩토리(EntityManagerFactory)**
  - **엔티티 매니저(EntityManager)**
- 3.2 영속성 컨텍스트란(persistence context)? 92p
- 3.3 엔티티 생명주기 92p
  - 1. 비영속(new/transient)
  - 2. 영속(managed)
  - 3. 준영속(detached)
  - 4. 삭제(removed)
- 3.4 영속성 컨텍스트의 특징 95p
  - 3.4.1 엔티티 조회
  - 1차 캐시
  - 3.4.2 엔티티 등록 99p
  - 3.4.3 엔티티 수정 102p
  - 3.4.4 엔티티 삭제 106p
- 3.5 플러시 107p
  - 3.5.1 플러시 모드 옵션 108p
- 3.6 준영속 109p
  - 3.6.1 엔티티를 준영속 상태로 전환 : detach()
  - 3.6.2 영속성 컨텍스트 초기화 : clear()
  - 3.6.3 영속성 컨텍스트 종료 : close()
  - 3.6.4 준영속 상태의 특징 114p
  - 3.6.5 병합 : merge() 115p

3.1 엔티티 매니저 팩토리와 앤티티 매니저

엔티티 매니저 팩토리(EntityManagerFactory)

- 엔티티 매니저를 만드는 공장
- 비용이 많이 든다.
- 한 개만 만들어서 애플리케이션 전체에서 공유하도록 설계되어 있다.
- **여러 스레드가 동시에 접근해도 안전하므로 서로 다른 스레드 간에 공유해도 된다.**

생성

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpabook");
```

- `Persistence.createEntityManagerFactory("jpabook")` 을 호출하면 META-INF/persistence.xml에 있는 정보를 바탕으로 EntityManagerFactory를 생성한다.

  <persistence.xml>

  ```
  <?xml version="1.0" encoding="UTF-8" ?>
  <persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.2">
  ```

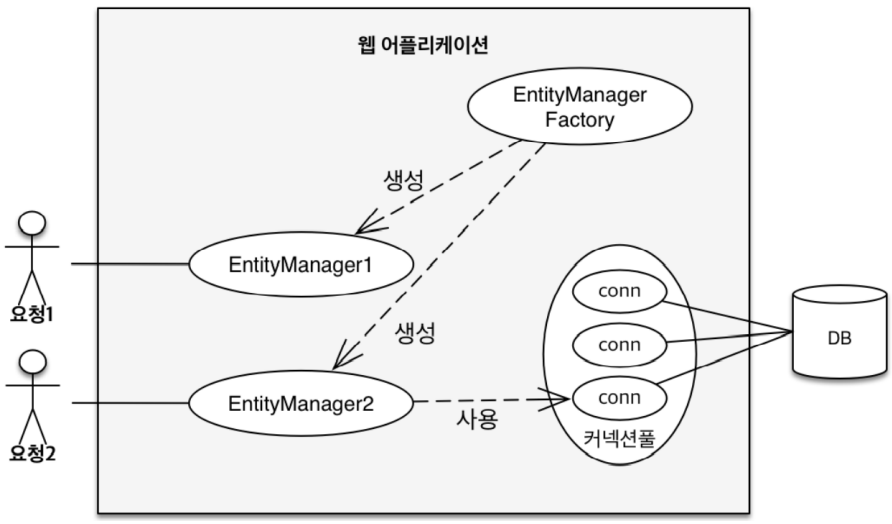
```
    <persistence-unit name="jpabook">
        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
            <property name="javax.persistence.jdbc.user" value="study_test"/>
            <property name="javax.persistence.jdbc.password" value=""/>
            <property name="javax.persistence.jdbc.url" value="jdbc:h2:tcp://localhost/~/study_test"/>

            생략
        </properties>
    </persistence-unit>
</persistence>
```

엔티티 매니저(EntityManager)

- 엔티티를 관리하는 관리자
- 엔티티를 저장, 수정, 삭제, 조회하는 등 엔티티와 관련된 모든 일을 처리
- 엔티티를 저장하는 가상의 데이터베이스라고 생각하면 됨
- **여러 스레드가 동시에 접근하면 동시성 문제가 발생하므로 스레드 간에 절대 공유하면 안 된다.**

생성

```
EntityManager em = emf.createEntityManager();
```



일반적인 웹 애플리케이션

3.2 영속성 컨텍스트란(persistence context)? 92p

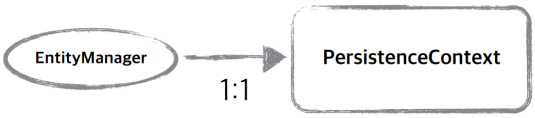
- 엔티티를 영구 저장하는 환경
- 엔티티 매니저를 생성할 때 하나 만들어진다.
- 엔티티 매니저를 통해서 영속성 컨텍스트에 접근하고 관리한다.

```
em.persist(member);
```

⇒ persist() 메소드는 엔티티 매니저를 사용해서 회원 엔티티를 영속성 컨텍스트에 저장한다.

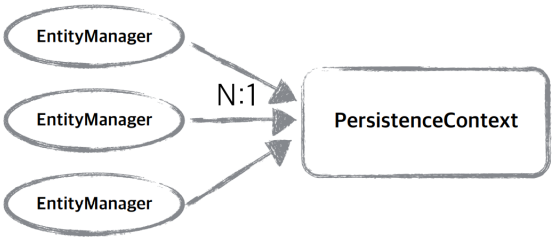
J2SE환경

- Java SE(Standard Edition)
- 엔티티 매니저와 영속성 컨텍스트가 1:1



J2EE, 스프링 프레임워크 같은 컨테이너 환경

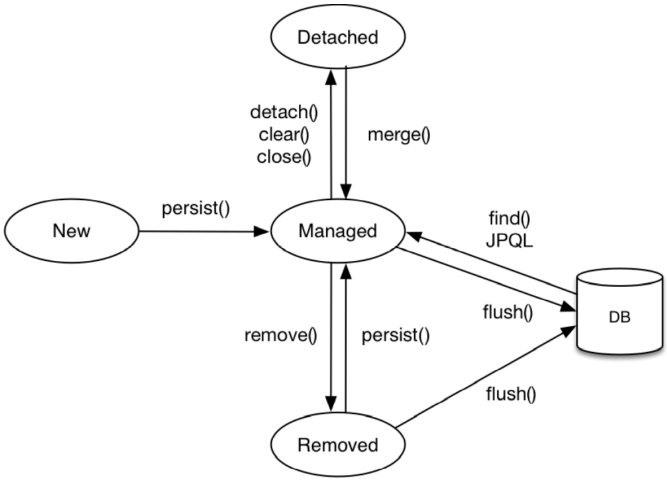
- Java EE(Enterprise Edition)
- 엔티티 매니저와 영속성 컨텍스트가 N:1



> J2SE, J2EE 차이
>
> - [자바 상식] J2EE, JDK, JRE, J2SE 차이 (tistory.com)
> - J2EE, JDK, JRE, J2SE 차이 (tistory.com)

(11장)

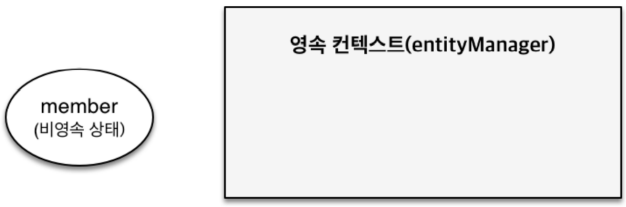
3.3 엔티티 생명주기 92p



1. 비영속(new/transient)

- 영속성 컨텍스트나 데이터베이스와는 전혀 관계가 없는 **새로운** 상태

```
// 객체를 생성한 상태(비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");
```



em.persist() 호출 전, 비영속 상태

2. 영속(managed)

- 영속성 컨텍스트에 저장된 상태. 즉 **관리**되는 상태
- **영속 상태**
  - 영속성 컨텍스트가 관리하는 엔티티
  - 즉 영속성 컨텍스트에 의해 관리된다는 의미

```
// 객체를 생성한 상태(비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");

EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

// 객체를 저장한 상태(영속)
em.persist(member);
```

em.persist() 호출 후, 영속 상태

3. 준영속(detached)

- 영속성 컨텍스트에 저장되었다가 **분리**된 상태
- 영속성 컨텍스트가 관리하던 영속 상태의 엔티티를 영속성 컨텍스트가 관리하지 않으면 준영속 상태가 된다.

```
// 회원 엔티티를 영속성 컨텍스트에서 분리, 준영속 상태
em.detach(member);
```

- 영속성 컨텍스트를 닫거나 초기화해도 영속 상태의 엔티티는 준영속 상태가 된다.

```
// 영속성 컨텍스트 닫기
em.close();
```

```
// 영속성 컨텍스트 초기화
em.clear();
```

4. 삭제(removed)

- **삭제**된 상태
- 엔티티를 영속성 컨텍스트와 데이터베이스에서 삭제한다.

```
// 객체를 삭제한 상태(삭제)
em.remove(member);
```

3.4 영속성 컨텍스트의 특징 95p

- 영속성 컨텍스트와 식별자 값
  - 영속성 컨텍스트는 엔티티를 식별자 값(@Id로 테이블의 기본 키와 매핑한 값)으로 구분한다.
  - 영속 상태는 식별자 값이 반드시 있어야 한다(없으면 예외 발생).
- 영속성 컨텍스트와 데이터베이스 저장
  - JPA는 보통 트랜잭션을 커밋하는 순간 영속성 컨텍스트에 새로 저장된 엔티티를 데이터베이스에 반영하는데 이것을 **플러시(flush)**라고 한다.
- 영속성 컨텍스트가 엔티티를 관리할 때 장점
  - 1차캐시
  - 동일성 보장
  - 트랜잭션을 지원하는 쓰기 지연
  - 변경 감지
  - 지연 로딩

3.4.1 엔티티 조회

1차 캐시

- 영속성 컨텍스트 내부에 가지고 있는 캐시
- 영속 상태의 엔티티는 모두 1차 캐시에 저장된다.

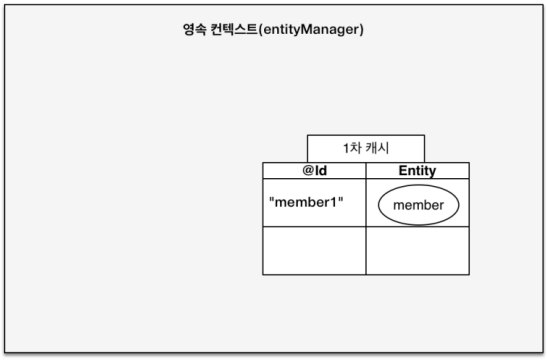
```
// 엔티티를 생성한 상태(비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");

// 엔티티를 영속 => 1차 캐시에 저장됨
em.persist(member);

// 1차 캐시에서 조회
Member member = em.find(Member.class, "member1");
```

1차 캐시에 회원 엔티티 저장

`em.persist(member);`

- 1차 캐시의 key는 @Id로 매핑한 식별자 , value는 엔티티 인스턴스이다.
- 식별자 값은 데이터베이스 기본 키와 매핑되어 있다.

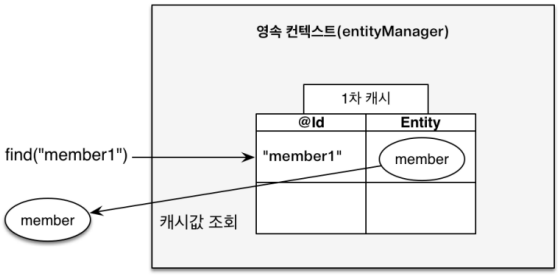
⇒ 영속성 컨텍스트에 데이터를 저장하고 조회하는 모든 기준은 데이터베이스 기본 키 값이다.



1차 캐시에서 엔티티 조회

```
Member member = em.find(Member.class, "member1");
```

- em.find()를 호출하면 1차 캐시에서 식별자 값으로 엔티티를 찾는다.
- 찾는 엔티티가 없으면 데이터베이스를 조회하지 않고 메모리에 있는 1차 캐시에서 엔티티를 조회한다.



▼ find()

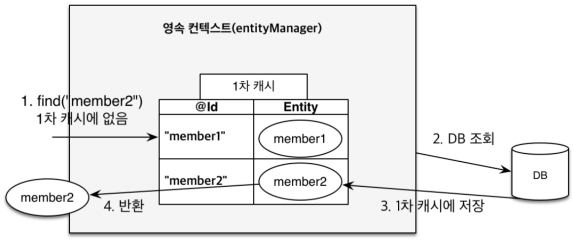
`em.find(엔티티 클래스의 타입, 조회할 엔티티의 식별자 값);`

```
// EntityManger.find() 메소드 정의
public <T> T find(Class<T> entityClass, Object primarykey);
```

데이터베이스에서 조회

```
Member member = em.find(Member.class, "member2");
```

1. en.find(Member.class, "member2") 실행
2. member2가 1차 캐시에 없으므로 데이터베이스에서 조회
3. 조회한 데이터로 member2 엔티티를 생성해서 1차 캐시에 저장(영속 상태)
4. 조회한 엔티티를 반환



영속 엔티티의 동일성 보장

- 영속성 컨텍스트는 성능상 이점과 엔티티의 동일성을 보장한다.

```
Member a = em.find(Member.class, "member1");
Member b = em.find(Member.class, "member1");

a == b; // 동일성 비교 true
```

동일성 비교(`a == b;`)

- em.find(Member.class, "member1")를 반복해서 호출해도 영속성 컨텍스트는 1차 캐시에 있는 같은 엔티티 인스턴스를 반환

- 따라서 a와 b는 같은 인스턴스고 결과는 true이다.

> 💡 **동일성과 동등성**
>
> - 동일성(identity) : **실제 인스턴스가 같다**. 따라서 참조 값을 비교하는 `==` 비교의 값이 같다.
> - 동등성(equality) : 실제 인스턴스는 다를 수 있지만 **인스턴스가 가지고 있는 값이 같다**. 자바에서 동등성 비교는 `equals()` 메소드를 구현해야 한다.

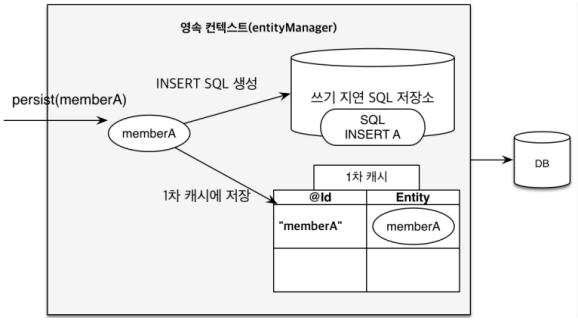
- 1차 캐시로 반복 가능한 읽기(REPEATABLE READ) 등급의 트랜잭션 격리 수준을 데이터베이스가 아닌 애플리케이션 차원에서 제공

3.4.2 엔티티 등록 99p

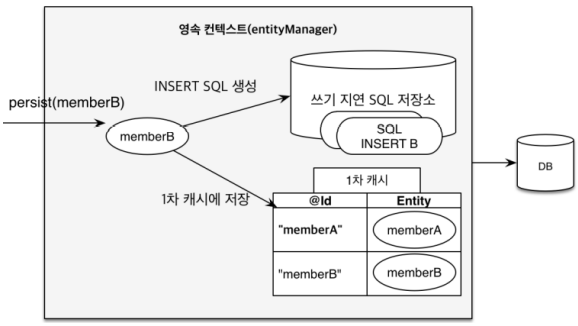
```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
// 엔티티 매니저는 데이터 변경시 트랜잭션을 시작해야 한다.
transaction.begin(); // [트랜잭션] 시작

em.persist(memberA);
em.persist(memberB);
// 여기까지 INSERT SQL을 데이터베이스에 보내지 않는다.

// 커밋하는 순간 데이터베이스에 INSERT SQL을 보낸다.
transaction.commit(); // [트랜잭션] 커밋
```

- 트랜잭션을 지원하는 쓰기지연(transactional write-behind)트랜잭션을 커밋할 때 모아둔 쿼리를 데이터베이스에 보낸다.

`em.persist(memberA);`

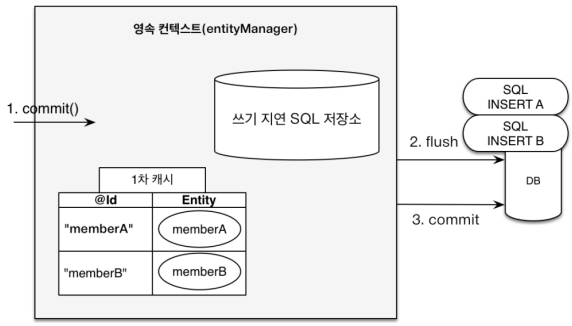


`em.persist(memberB);`



`transaction.commit();`

- 트랜잭션을 커밋하면 엔티티 매니저는 우선 영속성 컨텍스트를 플러시한다.
- 플러시 : 영속성 컨텍스트의 변경 내용을 데이터베이스에 동기화하는 작업



- 이때 등록, 수정, 삭제한 엔티티를 데이버테이스에 반영한다.

⇒ 쓰기 지연 SQL 저장소에 모인 쿼리를 데이터베이스에 보낸다.

3.4.3 엔티티 수정 102p

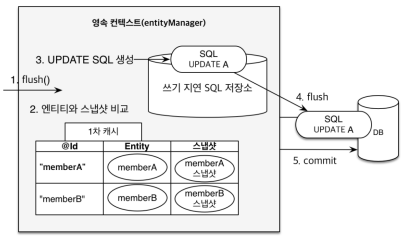
SQL 수정 쿼리의 문제점

- 수정 쿼리가 많아지고 비즈니스 로직을 분석하가 위해 SQL을 계속 확인해야한다.
- 결국 비즈니스 로직이 SQL에 의존하게 된다.

변경 감지(dirty checking)

- 엔티티의 변경사항을 데이터베이스에 자동으로 반영하는 기능

```
EntityManager em = emf.createEntityManager
EntityTransaction transaction = em.getTran
transaction.begin(); // [트랜잭션] 시작

// 영속 엔티티 조회
Member memberA = em.find(Member.class, "me

// 영속 엔티티 데이터 수정
memberA.setUsername("hi");
memberA.setAge(10);

//em.update(member) 이런 코드가 있어야 하지 않을까?

transaction.commit(); // [트랜잭션] 커밋
```



1. 트랜잭션을 커밋하면 엔티티 매니저 내부에서 먼저 플러시(flush())가 호출된다.
2. 엔티티와 스냅샷을 비교해서 변경된 엔티티를 찾는다.
3. 변경된 엔티티가 있으면 수정 쿼리를 생성해서 쓰기 지연 SQL 저장소에 보낸다.
4. 쓰기 지연 저장소의 SQL을 데이터베이스에 보낸다.
5. 데이터베이스 트랜잭션을 커밋한다.

- JPA는 엔티티를 영속성 컨텍스트에 보관할 때, 최초 상태를 복사해서 저장해두는데 이것을 **스냅샷**이라고 한다.
- 플러시 시점에 스냅샷과 엔티티를 비교해서 변경된 엔티티를 찾는다.
- **변경 감지는 영속성 컨텍스트가 관리하는 영속 상태의 엔티티에만 적용된다.**
- JPA의 기본 전략은 엔티티의 모든 필드를 업데이트 한다.
  - 필드가 많거나 저장되는 내용이 너무 크면 수정된 데이터만 사용해서 동적으로 UPDATE SQL을 생성하는 전략을 선택하면 된다.

    `@org.hibernate.annotations.DynamicUpdate`
  - 참고) 데이터를 저장할 때 데이터가 존재하는(null이 아닌) 필드로만 INSERT SQL을 동적으로 생성하는 @DynamicInsert 도 있다.

3.4.4 엔티티 삭제 106p

```
//삭제 대상 엔티티 조회
Member memberA = em.find(Member.class, “memberA");

em.remove(memberA); //엔티티 삭제
```

- 삭제 쿼리를 쓰기 지연 SQL 저장소에 등록한다.
- 이후 트랜잭션을 커밋해서 플러시를 호출하면 실제 데이터베이스에 삭제 쿼리를 전달한다.
- 참고) em.remove(memberA)를 호출하는 순간 memberA는 영속성 컨텍스트에서 제거된다.

3.5 플러시 107p

플러시(flush())

- **영속성 컨텍스트의 변경 내용을 데이터베이스에 반영**
- 플러시를 실행하면
  1. 변경 감지 동작 - 수정된 엔티티를 찾는다.

     수정된 엔티티는 수정 쿼리를 만들어 쓰기 지연 SQL 저장소에 등록한다.
  2. 쓰기 지연 SQL 저장소의 쿼리를 데이터베이스에 전송한다(등록, 수정, 삭제쿼리).
- 영속성 컨텍스트를 플러시하는 방법

  ▼ `em.flush()`**를 직접 호출**
  - 앤티티 매니저의 flush() 메소드를 직접 호출해서 영속성 컨텍스트를 강제로 플러시힌다.
  - 테스트나 다른 프레임워크와 JPA를 함께 사용할 때를 제외하고 거의 상용하지 x

  ▼ **트랜잭션 커밋 시 플러시 자동 호출**
  - 트랜잭션을 커밋하기 전에 플러시를 호출해서 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영해야 한다.

    ⇒ JPA는 트랜잭션으르 커밋할 때 플러시를 자동으로 호출한다.

  ▼ **JPQL 쿼리 실행 시 플러시 자동 호출**
  - JPQL이나 Criteria(10장 참고) 같은 객체지향 쿼리를 호출할 때 플러시가 자동 실행된다.

    ```
    em.persist(memberA); // 엔티티 memberA를 영속 상태로 만듦
    em.persist(memberB);
    em.persist(memberC);

    // 중간에 JPQL 실행
    query = em.createQuery("select m from Member m", Member.class);
    List<Member> members = query.getResultList();
    ```

  - memberA, memberB, memberC는 영속성 컨텍스트에는 있지만 데이터베이스에는 아직 반영되지 않았다.
  - 따라서 쿼리를 실행하기 직전에 영속성 컨텍스트를 플러시해서 변경 내용을 데이터베이스에 반영해야 한다.

    ⇒ JPA는 JPQL을 실행할 때 플러시를 자동 호출한다.
  - 참고) 식별자를 기준으로 조회하는 find() 메소드를 호출할 때는 플러시가 실행되지 않는다.

3.5.1 플러시 모드 옵션 108p

- 엔티티 매니저에 플러시 모드를 직접 지정하는 방법 : `javax.persistence.FlushModeTyupe` 사용
  - `FlushModeType.AUTO` : (트랜잭션)커밋이나 쿼리를 실행할 때 플러시(기본값)
  - `FLusModeType.COMMIT` : 커밋할 때만 플러시

```
em.setFlushMode(FlushModeType.COMMIT) // 플러시 모드 직접 설정
```

> 💡 플러시는
>
> - 영속성 컨텍스트를 비우지 않는다.
> - 영속성 컨텍스트의 변경내용을 데이터베이스에 동기화
> - 트랜잭션이라는 작업 단위가 중요 -> 커밋 직전에만 동기화하면 됨

3.6 준영속 109p

- 영속성 컨텍스트가 관리하는 영속 상태의 엔티티가 영속성 컨텍스트에서 분리(detached)된 것을 준영속 상태라고 한다.
- 즉 **영속 상태였다가 더는 영속성 컨텍스트가 관리하지 않는 상태를 준영속 상태**라 한다.
- 따라서 준영속 상태의 엔티티는 영속성 컨텍스트가 제공하는 기능을 사용할 수 없다.

> - 영속 상태 : 영속성 컨텍스트로부터 관리되는 상태
> - 준영속 상태 : 영속성 컨텍스트로부터 분리된 상태

영속 상태의 엔티티를 준영속 상태로 만드는 법

1. `em.detach(entity)` : 특정 엔티티만 준영속 상태로 전환
2. `em.clear()` : 영속성 컨텍스트를 완전히 초기화
3. `em.close()` : 영속성 컨텍스트를 종료

3.6.1 엔티티를 준영속 상태로 전환 : detach()

```
public void testDetached() {
    ...
    // 회원 엔티티 생성, 비영속 상태
    Member member = new Member();
    member.setId("memberA");
    member.setUsername("회원A");

    // 회원 엔티티 영속 상태
```

```
    em.persist(member);

    // 회원 엔티티를 영속성 컨텍스트에서 분리, 준영속 상태
    em.detach(member);

    transaction.commit(); // 트랜잭션 커밋
}
```

- detach() 메소드를 호출하는 순간 1차 캐시부터 쓰기 지연 SQL 저장소까지 해당 엔티티를 관리하기 위한 모든 정보가 제거된다. (그림)

3.6.2 영속성 컨텍스트 초기화 : clear()

- 영속성 컨텍스트를 초기화해서 해당 영속성 컨텍스트의 모든 엔티티를 준영속 상태로 만든다.

```
// 엔티티 조회, 영속 상태
Member member = em.find(Member.class, "memberA");

em.clear(); // 영속성 컨텍스트 초기화

// 준영속 상태
member.setUsername("changeName");
```

(그림)

- `em.clear()`로 인해 memberA는 영속성 컨텍스트가 관리하지 않으므로 준영속 상태이다.
- 준영속 상태이므로 영속성 컨텍스트가 지원하는 변경감지는 동작하지 않는다.
    - `member member.setUsername("changeName");` 회원의 이름을 변경해도 데이터베이스에 반영되지 않는다.

3.6.3 영속성 컨텍스트 종료 : close()

- 영속성 컨텍스트를 종료하면 해당 영속성 컨텍스트가 관리하던 영속 상태의 엔티티가 모두 준영속 상태가 된다.

```
public void closeEntityManager() {

    EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpabook");

    EntityManager em = emf.createEntityManager();
    EntityTransaction transaction = em.getTransaction();

    transaction.begin(); // 트랜잭션 - 시작

    Member memberA = em.find(Member.class, "memberA");
    Member memberB = em.find(Member.class, "memberB");

    transaction.commit(); // 트랜잭션 - 커밋

    em.close(); // 영속성 컨텍스트 닫기(종료)
}
```

(그림)

- 영속성 컨텍스트가 종료되어 더는 memberA, memberB가 관리되지 않는다.

3.6.4 준영속 상태의 특징 114p

▼ 거의 비영속 상태에 가깝다.

1차 캐시, 쓰기 지연, 변경 감지, 지연 로딩을 포함한 영속성 컨텍스트가 제공하는 어떠한 기능도 동작하지 않는다.

▼ 식별자 값을 가지고 있다.

비영속 상태는 식별자 값이 없을 수도 있지만

준영속 상태는 이미 한 번 영속 상태였으므로 반드시 식별자 값을 가지고 있다.

▼ 지연 로딩을 할 수 없다.

- 지연 로딩은 실제 객체 대신 프록시 객체를 로딩해두고 해당 객체를 실제 사용할 때 영속성 컨텍스트를 통해 데이터를 불러오는 방법이다.
- 준영속 상태는 영속성 컨텍스트가 더는 관리하지 않으므로 지연 로딩 시 문제가 발생한다.

3.6.5 병합 : merge() 115p

- 준영속 상태의 엔티티를 다시 영속 상태로 변경하려면 병합을 사용하면 된다.
- merge() 메소드는 준영속 상태의 엔티티를 받아서 그 정보로 새로운 영속 상태의 엔티티를 반환한다.

```
Member mergeMember = em.merge(member);
```

준영속 병합

```
public class ExamMergeMain() {
    //
    static EntityManagerFactory emf = Persistence.createEntityManagerFacotry("jpabook");

    public static void main(String args[]) {
        //
        Member member = createMember("memberA", "회원1");

        member.setUsername("회원명변경"); // * 3. 준영속 상태에서 변경

        mergeMember(member);
    }

    static Member createMember(String id, String username) {
        // == 영속성 컨텍스트1 시작 ==
        EntityManager em1 = emf.createEntityManager();
        EntityTransaction tx1 = em1.getTransaction();
        tx1.begin();

        Member member = new Member();
        member.setId(id);
        member.setUsername(username);

        em1.persist(member);
        tx1.commit();

        em1.close(); // 영속성 컨텍스트1 종료
        // member 엔티티는 준영속 상태가 됨.

        // == 영속성 컨텍스트1 종료 ==

        return member;
    }

    static void mergeMember(Member member) {
        // == 영속성 컨텍스트2 시작 ==
        EntityManager em2 = emf.createEntityManager();
        EntityTransaction tx2 = em2.getTransaction();

        tx2.begin();
        Member mergeMember = em2.merge(member); // mergeMember라는 영속성 컨텍스트2가 관리하는 새로운 영속 상태의 엔티티가 반환 // * 1, 2
        // member = em2.merge(member); // * 6
        tx2.commit(); // 데이터베이스에 반영됨

        // 준영속 상태
```

```
        System.out.println("member = " + member.getUsername()); // member = 회원명변경

        // 영속 상태
        System.out.println("mergeMember = " + mergeMember.getUsername()); // mergeMember = 회원명변경

        System.out.println("em2 contains member = " + em2.contains(member)); // em2 contains member = false // * 5
        System.out.println("em2 contains mergeMember = " + em2.contains(mergeMember)); // em contains mergeMember = true

        em2.close();
        // == 영속성 컨텍스트2 종료 ==
    }
}
```

*

1. merge() 실행
2. 파라미터로 넘어온 준영속 엔티티의 식별자 값으로 1차 캐시에서 엔티티를 조회한다(1차 캐시에 없으면 데이터베이스에서 조회해서 1차 캐시에 저장)
    - member의 식별자 값으로 조회한 결과는 "회원1"이다.
3. 조회한 영속 엔티티(mergeMember)에 member 엔티티의 값을 채워 넣는다.
    - member 엔티티의 값은 위 예제 3에서 set한 "회원명변경"을 mergeMember에 채워넣는다.
4. mergeMember 반환
    - mergeMember라는 새로운 영속 상태의 엔티티로 반환한다고 보면 됨.

5. 파라미터로 넘어온 엔티티는 병합 후에도 준영속 상태로 남아있다.
6. 따라서 준영속 엔티티를 참조하던 변수를 영속 엔티티를 참조하도록 변경하는 것이 안전

    `member = em2.merge(member);`

비영속 병합

- 병합merge은 비영속 엔티티도 영속 상태로 만들 수 있다.

```
Member member = new Member();
Member newMember = em.merge(member); // 비영속 병합
tx.commit();
```

- 병합은 준영속, 비영속을 신경 쓰지 않는다.
    - 식별자 값으로 엔티티를 조회할 수 있으면 불러서 병합하고

      조회할 수 없으면 새로 생성해서 병합한다.

      (save or update)