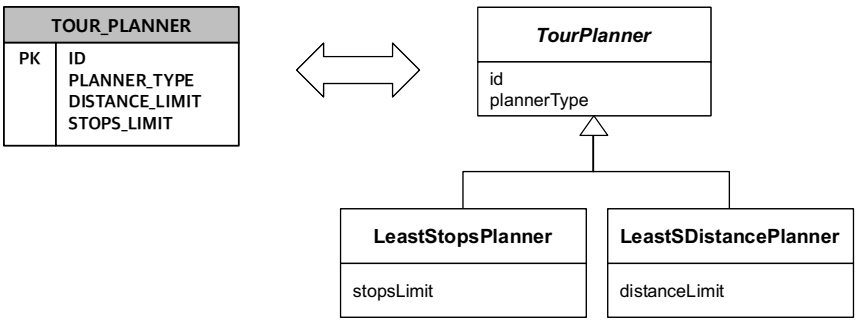
5 장 객체 경계

객체가 다른 객체에게 메시지를 전송하기 위해서는 먼저 객체에 접근해야 한다. 객체에 접근하는 가장 일반적인 메커니즘은 메시지를 전송할 객체 안에 메시지를 수신할 객체에 대한 참조를 인스턴스 변수의 형태로 포함시키는 것이다. 일반적으로 객체 참조의 값은 메모리에 생성된 객체의 주소가 된다. 객체는 클래스의 인스턴스 변수로 선언된 객체 참조를 통해 다른 객체로 이동할 수 있고, 이렇게 이동된 객체 역시 내부의 객체 참조를 통해 또 다른 객체로 이동할 수 있다. 따라서 객체지향 애플리케이션은 객체 참조로 연결된 거대한 객체 그래프의 형태를 띄게 된다.

문제는 필요한 객체가 항상 메모리 안에 존재하는 것은 아니라는 점이다. 시스템의 메모리는 제한적이기 때문에 경제적인 측면에서 객체가 실제로 필요해 지기 전까지는 상대적으로 느리지만 더 저렴한 외부 저장소에 객체를 저장하는 것이 일반적이다. 현재 객체를 저장하는데 사용되는 저장소의 주류는 관계형 데이터베이스 시스템(Relational DataBase Management System, RDBMS)이다.

객체지향 애플리케이션과 관계형 데이터베이스를 연동할 때 가장 큰 어려움은 객체와 테이블 사이의 **객체-관계 임피던스 불일치(Object-Relational Impedance Mismatch)** 문제를 해결해야 한다는 것이다. 객체-관계 임피던스 불일치 문제는 행동에 초점을 맞춘 객체 패러다임과 수학적인 원칙에 초점을 맞춘 관계형 패러다임 사이의 차이로 인해 객체와 데이터베이스 테이블이 구조적으로 서로 달라지는 문제를 말한다.

임피던스 불일치 문제의 대표적인 예는 테이블을 상속 계층에 매핑시키는 일이다. **4 장** 예제에서 TourPlanner 상속 계층과 TOUR_PLANNER 테이블 사이의 매핑이 이런 유형에 해당한다. 아래 그림에서 볼 수 있는 것처럼 데이터베이스는 관련된 데이터를 하나의 테이블 안에 모으는 것에 초점을 맞추는 반면 객체는 행동을 기반으로 타입을 분해하는 것에 초점을 맞추기 때문에 테이블과 객체의 정적인 구조가 달라지게 되는 것이다. 따라서 테이블을 읽어 컬럼의 값을 객체의 속성으로 설정하거나 객체의 속성을 테이블에 저장할 때 복잡한 변환과정을 거쳐야 한다.



관계형 패러다임과 객체 패러다임을 혼합할 때 마주치는 또 다른 어려움은 애플리케이션의 특정 기능을 구현하기 위해서는 전체 객체 그래프 중 일부만 필요하다는 것이다. 메모리의 크기는 제한적이기 때문에 데이터베이스에 저장된 객체 중에서 현재 기능 구현에 필요한 일부 객체들만

메모리로 로드해야 한다. 또한 데이터베이스는 본질적으로 애플리케이션의 전체 성능을 좌우할 정도로 느리기 때문에 성능 관점에서도 꼭 필요한 객체들만 조회하거나 저장해야 한다.

하지만 전체 그래프에서 필요한 객체만 조회한다는 게 말처럼 쉬운 일은 아니다. 연관관계로 연결된 객체들 중에서 조회를 시작할 객체와 조회를 끝난 객체를 결정해야 하고, 두 객체 사이에 존재하는 객체들도 같이 조회할 수 있어야 한다. 간단히 말해서 전체 객체 그래프 중에서 현재의 기능을 구현하는데 필요한 객체들만 포함하는 '경계'를 찾아 조회해야 한다는 것이다.

객체지향 커뮤니티에서 이 문제의 해결 방법으로 제시한 도구가 바로 ORM(Object-Relational Mapping, 객체 관계 매핑)이다. ORM 을 사용하면 데이터베이스가 아니라 객체를 기준으로 영속성 메커니즘을 처리할 수 있다. 또한 ORM 이 제공하는 영속성 전이나 지연 로딩과 같은 다양한 기법을 사용하면 객체 참조로 연결된 거대한 객체 그래프 안에서 원하는 경계 안의 객체들만 손쉽게 조회하거나 저장할 수 있다. 자바 커뮤니티에서의 ORM 표준 스펙을 JPA(Java Persistence API)라고 부른다.

이번 장에서는 여행 애플리케이션의 객체 경계 문제를 ORM 을 사용해서 해결하는 예제를 살펴볼 것이다. 의아할 수도 있겠지만 이번 장의 주제는 ORM 이 객체지향 애플리케이션의 경계 문제를 해결하는데 좋은 도구가 아니라는 점을 설명하는 것이다. 이번 장을 읽고 나면 ORM 을 잘못 사용하면 오히려 도메인 관심사와 데이터베이스 관심사의 경계가 모호해진다는 점을 이해하게 될 것이다.

[TODO – 박스 -> impedance mismatch 설명 추가]

ORM 을 제대로 사용하기 위해서는 ORM 의 구조 측면과 동작 측면 모두를 이해해야 한다. ORM 의 구조 측면은 데이터베이스 테이블과 컬럼을 클래스와 인스턴스 변수에 매핑하는 방법에 관한 것이다. ORM 의 동작 측면은 데이터베이스에 객체를 저장하거나 조회하는 방법에 관한 것이다. 마틴 파울러(Martin Fowler)가 지적한 것처럼 대부분의 사람들은 ORM 을 사용할 때 매핑을 다루는 구조 측면에 집중하는 경향이 있지만 ORM 에서 이해하기 어려운 부분은 객체를 저장하거나 조회하는 방식을 다루는 동작 측면이다[Fowler 2002].

이번 장에서는 먼저 여행 애플리케이션의 클래스를 테이블과 매핑하면서 ORM 의 구조 측면을 설명하기로 한다. 이어서 이전 장에서 애플리케이션의 기능을 구현한 코드를 살펴보면서 ORM 의 동작 측면을 살펴볼 것이다. 마지막으로 새로운 기능을 추가하면서 객체 경계를 고민하지 않을 때 발생할 수 있는 문제점들을 살펴보기로 하자.

앞으로 ORM 의 구조 측면과 동작 측면을 설명하기 위해 마틴 파울러가 [Fowler 2002]에서 제안한 패턴들의 이름을 이용할 것이다. ORM 의 구조 측면과 동작 측면을 깊이 있게 이해하고 싶다면 마틴 파울러의 책을 정독해 보기를 권한다.

다시 한번 강조하지만 이번 장의 목표는 ORM 을 깊이 있게 설명하는 것이 아니다. ORM 의 한계와 단점을 이해하는데 필요한 지식만을 얇게 살펴보는 것이다. ORM 이나 JPA 에 관해 더 자세히 알고 싶다면 [김영한 2015]를 참고하기 바란다.

ORM 의 구조 측면

객체와 데이터베이스 테이블의 가장 큰 차이점은 관계를 처리하는 방법이다. 객체는 다른 객체에 대한 참조를 보관하고 이 참조를 이용해서 다른 객체에 접근한다. 테이블은 외래 키(foreign key)라고 불리는 다른 테이블에 대한 키를 이용해서 다른 테이블에 접근한다.

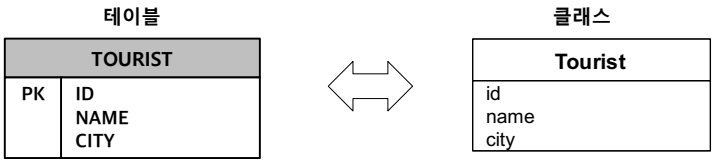
식별자 필드

객체와 테이블 사이의 관계에 대한 차이점을 해소하기 위해 먼저 ORM 은 테이블에 매핑되는 모든 클래스에 **식별자 필드(Identity Field)**를 포함시킨다. **식별자 필드**란 테이블의 기본 키(primary key)를 저장하기 위해 클래스 안에 선언된 인스턴스 변수를 말한다. 기본 키를 저장할 인스턴스 변수에 @Id 어노테이션을 추가하면 JPA 가 데이터를 조회하거나 저장할 때 이 인스턴스 변수를 기본 키로 사용한다. @GeneratedValue 는 테이블의 기본 키를 자동으로 생성할 때 사용하는 어노테이션으로 기본 키를 생성하는 전략을 지정할 수 있다. 예제에서는 기본 키 생성을 데이터베이스에 위임하는 GenerationType.IDENTITY 전략을 사용한다.

아래 Tourist 클래스의 경우 인스턴스 변수 id 가 TOURIST 테이블의 ID 컬럼에 매핑되는 식별자 필드에 해당한다.

```
@Entity
public class Tourist {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;
}
```

@Entity 어노테이션은 JPA 에게 이 클래스가 테이블에 매핑된다는 사실을 알려준다. 인스턴스 변수인 name 처럼 어노테이션을 추가하지 않으면 인스턴스 변수의 이름과 동일한 컬럼에 자동으로 매핑 된다. 따라서 Tourist 클래스의 인스턴스 변수 name 은 TOURIST 테이블의 NAME 컬럼에 매핑 된다. JPA 는 테이블 이름을 지정할 수 있는 @Tabel 어노테이션을 제공한다. Tourist 처럼 이 어노테이션을 생략하면 클래스의 이름과 동일한 TOURIST 테이블에 매핑된다.

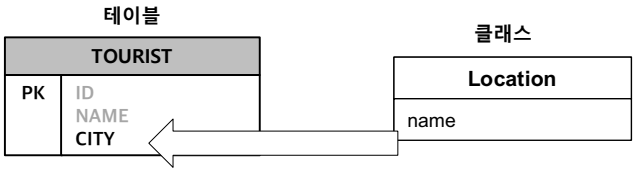


Tourist 클래스에서 알 수 있는 것처럼 클래스와 테이블을 매핑하는 작업은 대체로 지루하고 반복적인 작업이다. 따라서 매핑 규칙을 코드로 작성하는 대신 일반적인 ORM 은 예제처럼 클래스에 어노테이션을 추가하거나 외부의 XML 파일에 매핑 규칙을 선언적으로 정의한 후 외부의 도구가 자동으로 매핑을 해석하도록 하는 방식을 주로 사용한다. 이를 **메타데이터 매핑(Metadata Mapping)**이라고 부른다.

포함 값

객체지향 설계는 작은 크기의 객체를 선호하지만 데이터베이스 설계는 논리적으로 연관된 최대한 많은 속성들을 동일한 테이블에 모으려고 한다. 따라서 데이터베이스 테이블에 포함된 컬럼의 수가 객체에 포함된 속성의 수보다 많은 것이 일반적이다. 결과적으로 어떤 객체는 테이블에 직접 매핑 될 수도 있지만 어떤 객체는 테이블의 일부로 매핑 된다.

예를 들어 Tourist 클래스의 city 는 Location 클래스로 선언되지만 TOURIST 테이블에서는 CITY 라는 단일 컬럼에 저장된다. 다시 말해서 Location 전체가 TOURIST 테이블의 일부로 **포함**된다. 이처럼 어떤 클래스가 테이블에 매핑 되지 않고 테이블의 일부 컬럼으로 매핑 되는 방식을 **포함 값(Embedded Value)** 매핑이라고 부른다. 여기에서 '값'이라는 용어를 사용하는 이유는 Location 이 **값 객체(Value Object)**이기 때문이다. 값 객체에 대해서는 **XX 장**에서 자세히 살펴 볼 예정이다.



포함 값은 두 객체가 1:1 관계로 연결되는 경우에 자주 사용된다. 하지만 컬렉션에 포함된 원소의 개수가 적고 원소의 추가나 삭제가 빈번하게 일어나지 않는 경우에는 1:N 관계로 연결된 컬렉션의 요소로 사용될 수도 있다. 뒤에서는 포함 값을 컬렉션의 요소로 사용하는 예로 Path 매핑을 살펴볼 것이다.

JPA 에서는 @Embeddable 어노테이션을 이용해서 Location 클래스가 테이블의 일부로 포함된다는 사실을 정의한다.

```
@Embeddable
@Access(AccessType.FIELD)
public class Location {
  private String name;

  public static Location at(String name) {
    return new Location(name);
  }

  private Location(String name) {
    this.name = name;
  }
```

```
  protected Location() {
  }

  public String getName() {
    return name;
  }ㅌㅌ
}
```

Location 을 속성으로 포함하는 클래스에서는 해당 속성에 @Embedded 어노테이션을 추가해서 Location 의 속성이 테이블의 일부 컬럼에 매핑되도록 설정할 수 있다. Location 의 인스턴스 변수 name 을 다른 이름으로 정의하고 싶다면 @AttributeOverride 어노테이션을 추가해서 매핑될 컬럼의 이름을 변경할 수 있다. 다음 코드에서 Location 클래스의 name 속성은 NAME 이 아니라 CITY 라는 이름을 가진 컬럼에 매핑된다.

```
@Entity
public class Tourist {
  @Embedded
  @AttributeOverride(name="name", column = @Column(name="CITY"))
  private Location city;
}
```

외래 키 매핑

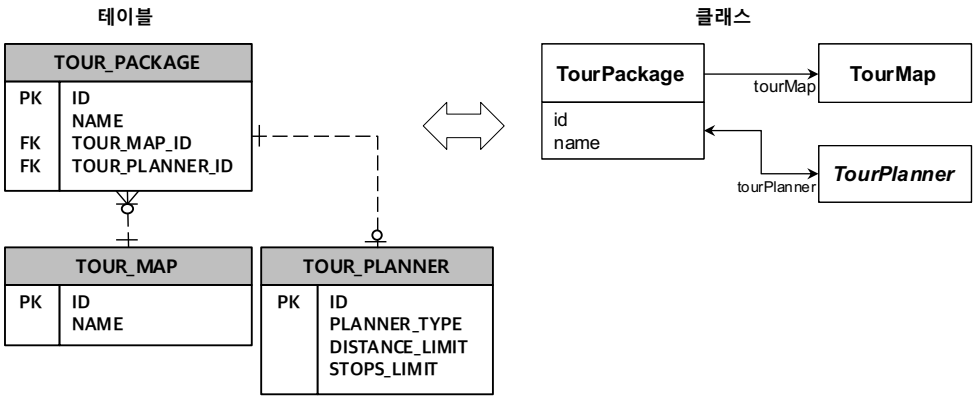
이제 TourPackage 를 매핑하자. TourPackage 는 TOUR_PACKAGE 테이블에 매핑되고 **식별자 필드**인 id 는 TOUR_PACKAGE 테이블의 ID 컬럼에, 인스턴스 변수 name 은 NAME 컬럼에 매핑된다. 여기까지는 Tourist 의 매핑 방법과 특별한 차이가 없다.

```
@Entity
public class TourPackage {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;
}
```

어려운 부분은 TOUR_PACKAGE 테이블과 TOUR_MAP, TOUR_PLANNER 사이의 테이블 관계를 객체 관계로 매핑하는 방법이다. 객체는 참조를 통해 연결된다. 테이블은 외래 키를 통해 연결된다. 따라서 객체 사이의 참조를 테이블 사이의 관계로 매핑하기 위해서는 객체 참조를 테이블의 외래 키와 매핑해야 한다. 이를 **외래 키 매핑(Foreign Key Mapping)**이라고 부른다.

TOUR_PACKAGE 테이블은 TOUR_MAP 테이블을 참조하는 외래 키 TOUR_MAP_ID 와 TOUR_PLANNER 테이블을 참조하는 외래 키 TOUR_PLANNER_ID 를 포함한다. TourPackage 클래스는 TourMap 인스턴스를 참조하는 객체 참조 tourMap 과 TourPlanner 인스턴스를 참조하는

객체 참조 tourPlanner 를 포함한다. 클래스 사이의 테이블 사이의 관계로 매핑하기 위해서는 **외래 키 매핑(Foreign Key Mapping)** 패턴에 따라 두 개의 객체 참조를 두 개의 외래 키에 매핑해야 한다.



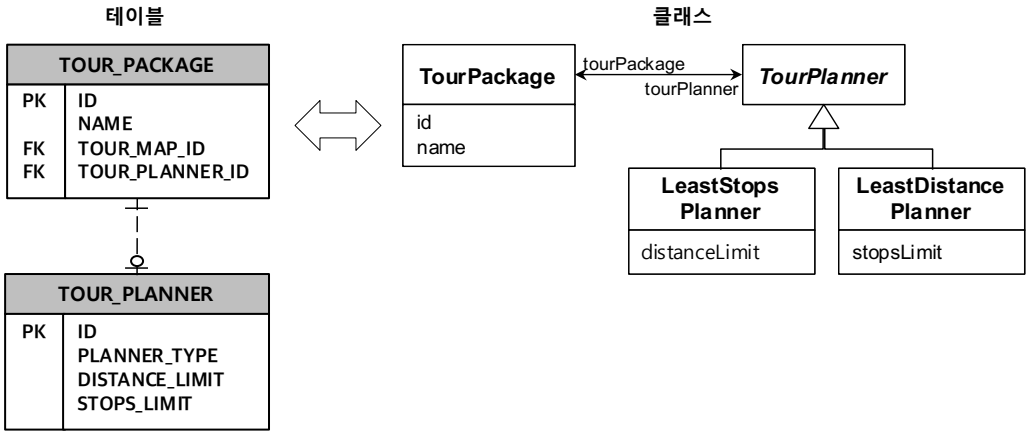
JPA 는 외래 키 매핑을 위해 두 가지 종류의 어노테이션을 제공한다. 하나는 다중성(multiplicity)을 정의하는 어노테이션이고, 다른 하나는 외래 키를 정의하는 어노테이션이다. 다중성을 정의하기 위해서는 @OneToOne, @OneToMany, @ManyToOne, @ManyToMany 어노테이션을 사용하고, 외래 키를 지정하기 위해서는 @JoinColumn 어노테이션을 사용한다.

TOUR_PACKAGE 테이블과 TOUR_MAP 테이블 사이의 다중성은 N:1 이기 때문에 @ManyToOne 어노테이션을 사용한다. 반면에 TOUR_PACKAGE 테이블과 TOUR_PLANNER 테이블 사이의 다중성은 1:1 이므로 @OneToOne 어노테이션을 사용한다. 만약 1:N 이나 N:N 관계를 지정하고 싶다면 @OneToMany 나 @ManyToMany 어노테이션을 사용하면 된다. 외래 키가 TOUR_MAP 과 TOUR_PLANNER 테이블을 참조하는 두 외래 키 TOUR_MAP_ID와 TOUR_PLANNER_ID를 지정하기 위해 @JoinColumn 어노테이션을 추가했다.

```
@Entity
public class TourPackage {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;

  @ManyToOne
  @JoinColumn(name="TOUR_MAP_ID")
  private TourMap tourMap;

  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name="TOUR_PLANNER_ID")
  private TourPlanner tourPlanner;
}
```

상속 매핑

TourPlanner 를 TOUR_PLANNER 테이블에 매핑하기 위해서는 두 가지 이슈를 해결해야 한다. 첫 번째 이슈는 상속 관계를 테이블에 매핑하는 것이고, 두 번째 이슈는 TourPlanner 와 TourPackage 사이의 양방향 연관관계를 매핑하는 것이다.



먼저 TourPlanner 상속 계층을 테이블에 매핑하는 방법을 살펴보자. ORM 을 이용해서 상속 계층을 테이블로 매핑하는 방식에는 **단일 테이블 상속(Single Table Inheritance)**, **클래스 테이블 상속(Class Table Inheritance)**, **구체 테이블 상속(Concrete Table Inheritance)**이 있다. 예제에서는 하나의 테이블인 TOUR_PLANNER 에 매핑하는 **단일 테이블 상속** 방식을 사용하기로 한다.

상속 계층에 속하는 여러 클래스들을 하나의 테이블에 저장하고 구분하기 위해서는 매핑할 구체 클래스의 타입을 저장할 컬럼을 추가해야 한다. TOUR_PLANNER 테이블에서는 PLANNER_TYPE 컬럼에 클래스 타입을 할당한다. 여기에서는 PLANNER_TYPE 컬럼의 값이 'LEAST_STOPS'이면 LeastStopsPlanner 클래스에 매핑하고, 'LEAST_DISTANCE'이면 LeastDistancePlanner 클래스에 매핑하기로 하자.

JPA 에서 상속 계층을 매핑하기 위해서는 부모 클래스와 자식 클래스에 어노테이션을 추가해야 한다. 부모 클래스에는 @Inheritance 와 @DiscriminatorColumn 어노테이션을 추가하고, @Inheritance 에는 상속 방식을, @DiscriminatorColumn 에는 타입을 저장할 컬럼의 이름을 할당한다. 자식 클래스에는 @DiscriminatorValue 어노테이션을 추가하고 타입 컬럼의 값을 할당한다.

예제에서는 @Inheritance 의 값으로 단일 테이블 상속 방식을 나타내는 InheritanceType.SINGLE_TABLE 을, @DiscriminatorColumn 의 값으로 PLANNER_TYPE 을 할당했다. 또한 자식 클래스인 LeastStopsPlanner 와 LeastDistancePlanner 클래스 각각의 @DiscriminatorValue 에는 'LEAST_DISTANCE'와 'LEAST_STOPS'를 할당했다.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "PLANNER_TYPE")
public abstract class TourPlanner {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
  private Long id;

  @OneToOne(mappedBy = "tourPlanner")
  private TourPackage tourPackage;
}

@Entity
@DiscriminatorValue("LEAST_DISTANCE")
public class LeastDistancePlanner extends TourPlanner {
  private double distanceLimit;
}

@Entity
@DiscriminatorValue("LEAST_STOPS")
public class LeastStopsPlanner extends TourPlanner {
  private int stopsLimit;
}
```

연관관계 소유자

TourPlanner 클래스를 자세히 살펴보면 tourPackage 속성에 추가한 @OneToOne 의 mappedBy 속성 값이 "tourPlenner"라는 사실을 알 수 있다. mappedBy 속성은 객체 사이의 양방향 연관관계를 동기화하기 위해 설정하는 속성으로, 이 속성의 의미를 이해하기 위해서는 테이블에서 이야기하는 양방향 관계와 객체지향에서 이야기하는 양방향 관계의 의미가 다르다는 사실과 **연관관계 소유자(owner)**라는 개념을 이해해야 한다.

테이블에는 방향성이라는 개념이 없다. TOUR_PACKAGE 테이블에 TOUR_PLANNER 테이블에 대한 외래 키가 존재할 경우 테이블 조인(join)을 통해 TOUR_PACKAGE 에서 TOUR_PLANNER 방향으로 데이터를 조회하는 것도 가능하고 TOUR_PLANNER 에서 TOUR_PACKAGE 방향으로 데이터를 조회하는 것도 가능하다. 다시 말해서 데이터베이스에서는 어느 한쪽에 외래 키가 존재하면 어떤 방향으로도 조회가 가능하기 때문에 조회 측면에서는 양방향이라고 이야기할 수 있다.

반면에 객체의 참조는 항상 단방향이다. TourPackage 에서 TourPlanner 를 가리키는 객체 참조가 존재할 경우 TourPackage 에서 TourPlanner 의 방향으로 이동은 가능하지만 그 반대 방향으로의 이동은 불가능하다. 객체지향에서 두 객체를 양방향으로 연결하기 위해서는 두 개의 독립적인 객체 참조를 각 클래스에 추가하고 항상 서로를 가리키도록 동기화시키는 코드를 구현해야 한다.

이미 **4 장**에서 TourPackage 와 TourPlanner 클래스 사이의 양방향 연관관계를 동기화시키는 방법을 살펴본 적이 있다. 두 객체 참조를 동기화 시키기 위해서는 한 쪽의 참조가 설정될 때마다 반대쪽의 setter 메서드도 같이 호출해서 두 객체가 항상 서로를 가리키도록 해야 한다. 또한 setter 메서드가 반복적으로 호출되는 것을 막기 위해 현재의 참조와 인자로 전달된 참조가 동일한지를 체크해서 불필요한 메소드 호출을 방지해야 한다. TourPackageDAO 와 TourPlannerDAO 는 객체를 로드할 때마다 setter 메서드를 호출해서 객체 참조를 동기화시켰다는 점을 기억하라.

```
public class TourPackage {
```

```
  private TourPlanner tourPlanner;

  public TourPackage(Long id, String name, TourMap tourMap,
      TourPlanner tourPlanner) {
    ...
    setTourPlanner(tourPlanner);
  }

  public void setTourPlanner(TourPlanner tourPlanner) {
    if (this.tourPlanner == tourPlanner) {
      return;
    }

    this.tourPlanner = tourPlanner;
    this.tourPlanner.setTourPackage(this);
  }
}

public abstract class TourPlanner {
  private TourPackage tourPackage;

  public void setTourPackage(TourPackage tourPackage) {
    if (this.tourPackage == tourPackage) {
      return;
    }

    this.tourPackage = tourPackage;
    this.tourPackage.setTourPlanner(this);
  }
}
```

ORM 을 사용하면 양방향 연관관계의 동기화를 매우 간단하게 처리할 수 있다. 적절한 매핑 정보만 지정해 주면 동기화에 필요한 작업들을 ORM 이 알아서 처리해주기 때문이다. 이 때 고려해야 하는 것이 바로 연관관계 소유자라는 개념이다. 연관관계 소유자란 양방향 연관관계를 구성하는 두 객체 참조 중에서 동기화를 책임지는 객체를 말한다. 연관관계 소유자는 일반적으로 외래 키를 소유한 테이블과 매핑되는 클래스로 정한다. 연관관계 소유자쪽의 객체 참조가 설정될 때마다 반대쪽의 객체 참조가 연관관계 소유자를 가리키도록 동기화시켜야 하는데 이 때 사용하는 것이 바로 mappedBy 속성이다.

복잡해 보이지만 코드로 살펴보면 간단하다. 아래 코드는 위에서 살펴본 setter 코드와 개념저적으로 동일한 작업을 수행하도록 JPA 설정을 추가한 것이다.

```
@Entity
public class TourPackage {
  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name="TOUR_PLANNER_ID")
  private TourPlanner tourPlanner;
}

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "PLANNER_TYPE")
public abstract class TourPlanner {
```

```
  @OneToOne(mappedBy = "tourPlanner")
  private TourPackage tourPackage;
}
```

TOUR_PACKAGE 쪽에 TOUR_PLANNER 테이블을 가리키는 외래 키가 존재하기 때문에 연관관계 소유자는 TourPackage 가 된다. TourPackage 의 tourPlanner 참조에 @JounColumn 을 이용해서 외래키를 지정했다는 점에 주목하라. JPA 는 데이터베이스로부터 TourPlackage 를 로드할 때마다 외래키 TOUR_PLANNER_ID 에 해당하는 TourPlanner 를 로드한 후 객체 참조 tourPlanner 에 설정한다.

우리의 목적은 두 객체 참조를 동기화시키는 것이므로 TourPackage 의 tourPlanner 가 설정될 때마다 반대쪽인 TourPlanner 의 tourPackage 가 이 tourPlanner 를 가리키도록 JPA 에게 알려줘야 한다. 이때 사용하는 것이 바로 mappedBy 속성으로 값으로 "tourPlanner"를 지정하면 TourPackage 의 tourPlanner 에 해당 TourPlanner 인스턴스가 설정될 때마다 인스턴스 변수인 tourPackage 가 tourPlanner 를 가리키도록 자동으로 설정된다.

포함 값 컬렉션

다음으로 TourMap 클래스와 TOUR_MAP 테이블의 매핑 설정을 살펴보자. TourMap 클래스를 매핑하기 위해서는 먼저 내부에 리스트로 포함된 Path 를 TOUR_MAP_PATH 테이블에 매핑해야 한다.

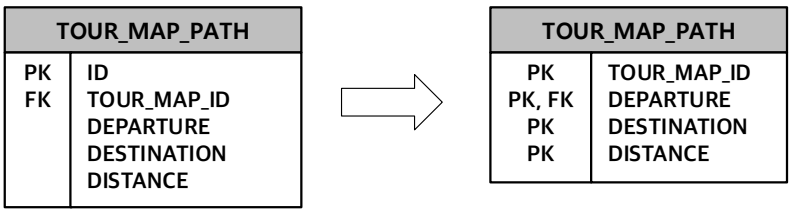
```
public class TourMap {
  ...
  private List<Path> paths = new ArrayList<>();
}
```

여기에서 문제는 Tour 클래스에서도 여행 경로를 표현하기 위해 Path 클래스를 재사용한다는 점이다.

```
public class TourMap {
  ...
  private List<Path> paths = new ArrayList<>();
}
```

TourMap 의 입장에서 Path 는 TOUR_MAP_PATH 테이블로 매핑돼야 한다. 하지만 Tour 의 입장에서는 Path 가 TOUR_PATH 테이블로 매핑돼야 한다. Path 는 다양한 클래스에서 재사용되고 여러 테이블에 매핑될 목적으로 설계됐기 때문에, 어노테이션을 선언하는 시점에 Path 클래스에 매핑될 특정한 테이블을 지정하는 것이 불가능하다. 이 문제를 해결할 수 있는 여러가지 방법이 있지만 여기에서는 Path 를 **포함 값**으로 설계하고, TourMap 과 Tour 에 포함 값의 컬렉션 형태로 포함시키는 방법을 사용하기로 한다.

포함 값의 컬렉션을 매핑할 때는 두 가지 기억할 사항을 염두에 둬야 한다. 첫 번째는 포함 값의 컬렉션을 매핑할 때는 클래스의 모든 속성이 매핑되는 테이블의 기본 키를 구성하도록 테이블을 설계해야 한다는 점이다. 기존의 ID 컬럼을 기본 키로 가지던 TOUR_MAP_PATH 테이블의 스키마를 ID 를 삭제하고 TOUR_MAP_ID, DEPARTURE, DESTINATION, DISTANCE 모두를 기본 키로 포함하도록 변경해야 한다.



포함 값의 컬렉션에 대해 기억해야 하는 두 번째 이슈는 한 요소가 수정되더라도 컬렉션에 포함된 모든 요소를 전부 삭제하고 다시 인서트한다는 것이다. 따라서 포함 값의 컬렉션은 가급적 컬렉션에 포함된 요소의 수가 적고 잘 변하지 않는 경우에 사용하는 것이 효과적이다.

앞에서 살펴본 Location 의 매핑과 동일한 방식으로 @Embeddable 을 추가하면 Path 를 **포함 값** 형태로 매핑할 수 있다.

```
@Embeddable
@Access(AccessType.FIELD)
public class Path {
  @AttributeOverride(name = "name", column = @Column(name="DEPARTURE"))
  private Location from;

  @AttributeOverride(name = "name", column = @Column(name="DESTINATION"))
  private Location to;

  private double distance;
}
```

JPA 에서 **포함 값**의 컬렉션을 매핑하기 위해서는 @ElementCollection 어노테이션을 사용한다. @CollectionTable 을 추가해서 Path 가 매핑될 테이블을 명시적으로 지정할 수 있다. 아래 코드에서 TourMap 에 포함된 Path 는 TOUR_MAP_PATH 테이블로 매핑된다. @CollectionTable 에 jponColumns 를 이용해서 TOUR_MAP_PATH 테이블에서 TOUR_MAP 테이블에 대한 외래 키로 사용할 컬럼을 지정하는 부분을 눈 여겨 보기 바란다. Path 가 다양한 테이블에 매핑돼야 하기 때문에 테이블의 외래 키 역시 Path 를 사용하는 클래스 문맥 안에서 선언해야 한다.

```
@Entity
public class TourMap {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;
```

```
  @ElementCollection
  @CollectionTable(name="TOUR_MAP_PATH",
      joinColumns = @JoinColumn(name="TOUR_MAP_ID"))
  private List<Path> paths = new ArrayList<>();
}
```

Tour 클래스에서 동일한 방법을 이용해서 Path 를 TOUR_PATH 테이블에 매핑한다.

```
@Entity
@ToString
public class Tour {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToOne
  @JoinColumn(name="TOURIST_ID")
  private Tourist tourist;

  @ManyToOne
  @JoinColumn(name="TOUR_PACKAGE_ID")
  private TourPackage tourPackage;

  @ElementCollection
  @CollectionTable(name="TOUR_PATH",
      joinColumns = @JoinColumn(name="TOUR_ID"))
  private List<Path> paths = new ArrayList<>();
}
```

이제 TourPackage 를 매핑하는 작업만 남았다. 지금까지 설명한 내용을 이해했다면 TourPackage 의 설정을 이해하는데 무리가 없을 것이다.

```
@Entity
public class TourPackage {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;

  @ManyToOne
  @JoinColumn(name="TOUR_MAP_ID")
  private TourMap tourMap;

  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name="TOUR_PLANNER_ID")
  private TourPlanner tourPlanner;
}
```

ORM 의 동작 측면

작업 단위

ORM 은 데이터베이스에 영향을 미치는 모든 내역을 추적한 후 트랜잭션을 커밋(commit)할 때 자동으로 수정된 내역을 데이터베이스에 반영한다. 예를 들어 트랜잭션을 시작하고, 데이터베이스로부터 Tourist 객체를 로드한 후, Tourist 객체의 속성을 변경하고, 트랜잭션을 커밋하면 ORM 은 Tourist 의 상태가 변경됐다는 사실을 감지하고 자동으로 TOURIST 테이블을 업데이트한다. 이처럼 변경 내역을 추적하고 자동으로 데이터베이스에 반영하기 위해 ORM 에 구현된 컴포넌트를 **작업 단위(Unit of Work)**라고 부른다.

작업 단위는 현재 트랜잭션 안에서 수정된 객체들의 집합을 유지한다. 트랜잭션 안에서 수정된 객체들을 가리켜 더티(dirty) 상태에 있다고 말한다. ORM 은 트랜잭션이 커밋될 때 더티 상태에 있는 객체들을 식별한 후 수정된 내역을 반영할 쿼리를 자동으로 생성하고 실행한다. 이처럼 더티 상태에 있는 객체들을 자동으로 데이터베이스에 저장하는 기능을 **자동 더티 체킹(automatic dirty checking)**이라고 부른다.

ORM 은 동일 트랜잭션 내에서 동일한 객체들이 여러 번 로드되는 것을 방지하기 위해 **식별자 맵(Identity Map)**을 사용한다. 식별자 맵을 데이터베이스의 기본키를 키로 저장하고 로드된 객체를 값으로 가지는 Map 이라고 생각해도 무방하다. ORM 은 객체를 로드하라는 요청을 받으면 데이터베이스로 요청을 보내기 전에 먼저 요청된 객체의 기본 키가 식별자 맵에 존재하는지 조사한다. 기본 키가 존재하지 않으면 데이터베이스로부터 객체를 로드한 후 식별자 맵에 기본 키와 객체를 추가한다. 기본 키가 이미 존재하는 경우에는 데이터베이스에 요청을 보내지 않고 식별자 맵을 이용해서 이미 로드된 객체를 찾아 반환한다. 따라서 식별자 맵을 사용하면 객체들을 캐싱하는 효과를 얻을 수 있기 때문에 성능이 향상될 수 있고, ORM 레벨에서 반복 읽기(REPEATABLE READ) 수준의 트랜잭션 격리 레벨을 제공할 수 있다.

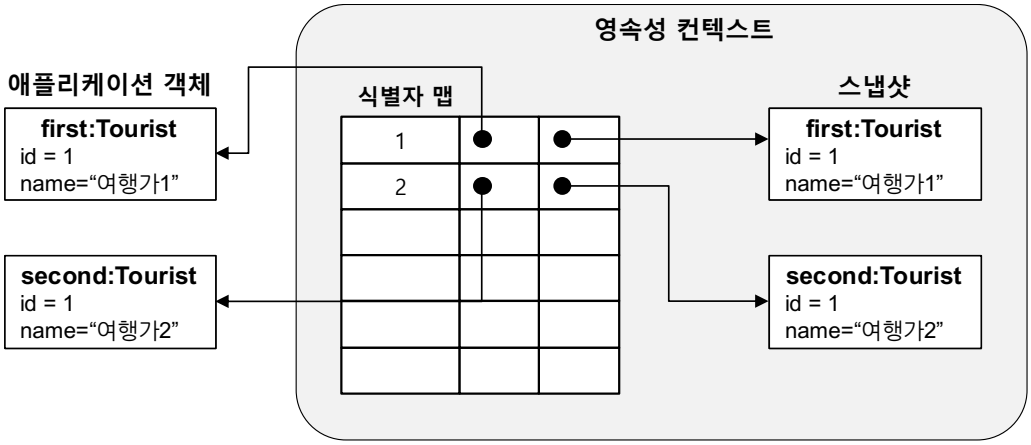
일반적으로 식별자 맵은 **작업 단위** 안에 위치한다. ORM 은 객체의 필드가 수정되거나 객체가 삭제되더라도 변경 내역을 즉시 데이터베이스에 반영하지 않는다. 트랜잭션 안에서 발생한 변경 내역을 추적한 후 트랜잭션이 커밋될 때 한번에 데이터베이스에 반영한다. 이를 **트랜잭션을 지원하는 쓰기 지연(transactional write-behind)**이라고 부른다. 작업 단위는 트랜잭션 커밋 시점에 식별자 맵에 저장된 모든 객체들의 변경 상태를 확인한 후 외래 키 제약 조건을 위반하지 않도록 SQL 문을 생성한다.

JPA 에서 **작업 단위**의 역할은 **영속성 컨텍스트(Persistence Context)**가 수행한다. 식별자 맵 역시 영속성 컨텍스트 안에 위치한다. 영속성 컨텍스트는 JPA 에서 제공하는 EntityManager 라는 객체 안에 생성되며, 항상 그런 것은 아니지만 일반적으로 트랜잭션을 시작할 때 생성되고 트랜잭션을 커밋하거나 롤백할 때 제거된다. 일단 영속성 컨텍스트가 생성되면 그 이후로 EntityManager 를 통해 조회한 모든 객체들의 복사본이 영속성 컨텍스트 안의 식별자 맵에 저장된다. 객체를 수정하고 트랜잭션을 커밋하면 작업 단위 역할을 맡은 영속성 컨텍스트가 식별자 맵에 보관된 객체의 상태를 확인한 후 수정된 객체들을 식별하고 SQL 문을 생성해서 데이터베이스에 변경내역을 반영한다.

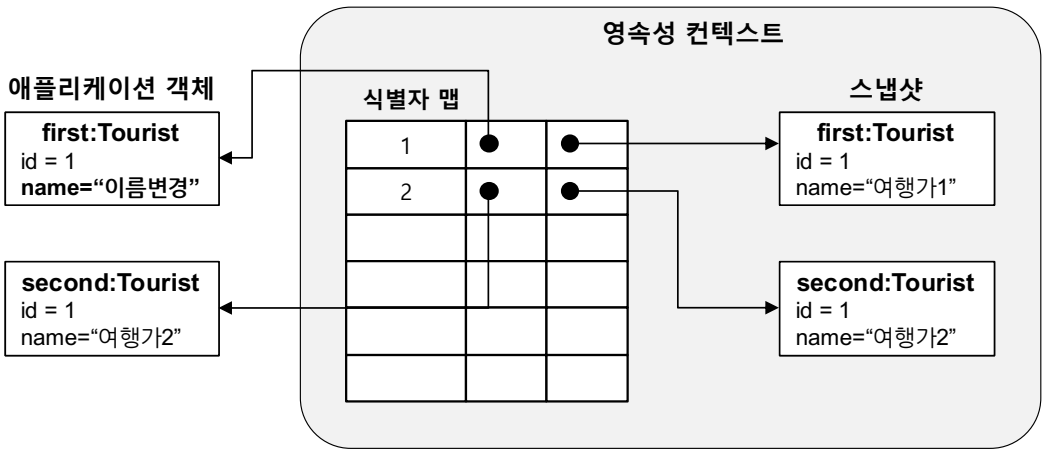
다음은 EntityManager 를 이용해서 Tourist 객체의 상태를 변경한 후 데이터베이스에 반영하는 코드를 나타낸 것이다. 먼저 EntityManager 의 getTransaction 메서드를 통해 트랜잭션을 생성한 후 begin 메서드를 이용해 트랜잭션을 시작한다. 기본 키가 각각 1 과 2 인 두 개의 Tourist 객체를 데이터베이스로부터 로드한 후 첫번째 Tourist 객체인 first 의 name 속성만 변경하고 트랜잭션을 커밋한다. first 객체의 상태가 변경됐기 때문에 기본 키가 1 인 first 의 레코드는 수정되고, 기본 키가 2 인 second 의 레코드는 실행되지 않을 것이다.

```
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();

Tourist first = entityManager.find(Tourist.class, 1L);
Tourist second = entityManager.find(Tourist.class, 2L);
first.changeName("새로운 이름");

transaction.commit();
```

앞에서 설명한 것처럼 JPA 는 first 와 second 객체를 데이터베이스에서 로드하자마자 로드 시점에 객체의 복사본인 스냅샷을 생성한 후 영속성 컨텍스트의 식별자 맵에 객체의 스냅샷을 추가한다.



애플리케이션에 반환된 first 의 name 속성을 "새로운 이름"으로 변경하더라도 영속성 컨텍스트에 저장된 스냅샷은 변경되지 않는다.



마지막으로 트랜잭션을 커밋하면 JPA 구현체는 영속성 컨텍스트에 저장된 객체들의 스냅샷과 애플리케이션 영역의 객체 상태를 비교한다. 위 그림에서는 first 객체의 상태만 변경됐고 second 객체의 상태는 변경되지 않았기 때문에 first 객체에 대한 업데이트 쿼리만 생성한다.

코드에서 EntityManager 에 대해 명시적으로 업데이트 요청을 보내는 문장이 없다는 점에 주목하라. 영속성 컨텍스트가 트랜잭션 커밋 시점에 자동으로 변경된 부분을 식별할 수 있기 때문에 객체 단위의 업데이트문을 실행할 필요가 없기 때문이다.

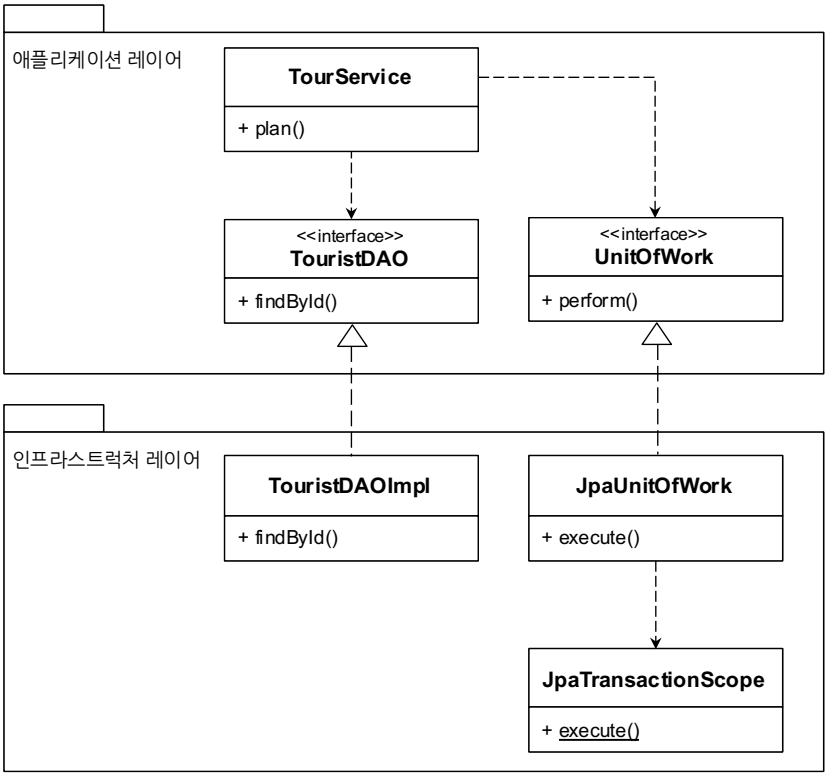
여행 애플리케이션에서 EntityManager 를 다루는 로직의 대부분은 모든 DAO 클래스의 기반 클래스인 JpaDAO 에 캡슐화되어 있다. JpaDAO 는 기존의 BaseDAO 와 유사하지만 직접 SQL 문을 실행하는 대신 EntityManager 를 이용해서 객체를 저장하거나 조회하는 기본 메서드를 제공한다.

다음은 JpaDAO 클래스의 기본 구현을 나타낸 것이다. EntityManagers 는 현재의 EntityManager 를 제공하는 헬퍼 크래스다.

```
public abstract class JpaDAO<T> {
  private final Class<T> entityClass =
    (Class<T>)((ParameterizedType) getClass().getGenericSuperclass())
       .getActualTypeArguments()[0];

  protected void saveOne(T entity) {
    EntityManagers.current().persist(entity);
   }

   protected T findOne(Long id) {
     return EntityManagers.current().find(entityClass, id);
   }

   protected T findOne(String jpql, Object [] params) {
     List<T> result = findMany(jpql, params);
     return !result.isEmpty() ? result.get(0) : null;
   }

   protected List<T> findMany(String jpql, Object [] params) {
     TypedQuery<T> query = EntityManagers.current()
                                .createQuery(jpql, entityClass);
      for(int current=0; current<params.length; current++) {
        query.setParameter(current+1, params[current]);
      }

      return query.getResultList();
   }
}
```

UnitOfWork 인터페이스의 구현 클래스로 지금까지 사용한 JdbcUnitOfWork 대신 JPA 에서 제공하는 EntityManager 를 이용해서 구현한 JpaUnitOfWork 를 주입해서 사용한다. **4 장**에서 인프라스트럭처 레이어가 애플리케이션 레이어에 의존하도록 의존성을 역전시켰기 때문에 새로운 JpaUnitOfWork 를 사용하더라도 애플리케이션 레이어의 코드를 전혀 변경할 필요가 없었다는 사실에 주목하라.

또한 JDBC 에서 JPA 로 영속성 기술을 변경했음에도 애플리케이션 레이어에 속한 DAO 인터페이스들은 그대로 유지한 채 인프라스트럭처 레이어의 구현 클래스들만 수정했다는 사실에도 주목하라. 인프라스트럭처 관련 기술을 변경했지만 더 핵심적인 관심사를 다루는 애플리케이션 레이어에는 어떤 수정도 하지 않았다는 점이 중요하다. 이 예제는 관심사를 분리하고 의존성의 방향을 제어하는 일이 코드 수정 관점에서 얼마나 중요한 지를 다시 한번 보여준다.



지연 로딩

ORM 의 입장에서 객체를 로드하라는 요청을 받았을 때 어떤 객체를 함께 로드해야 하는지 예상할 수 없다. 사용하지 않을 객체 그래프 전체를 메모리에 로딩하는 것은 비효율적이며 성능 관점에서도 문제가 발생할 수 밖에 없다. 따라서 ORM 은 최초에 필요한 객체만 로드하고 나머지 객체들은 연관 관계를 통해 접근되는 시점에 자동으로 로딩하는 기능을 지원하는데 이를 **지연 로딩(Lazy Loading)**이라고 부른다. 반대로 어떤 객체를 로드할 때 연관된 객체를 항상 함께 로드하는 것을 **즉시 로딩(Eager Loading)**이라고 부른다.

JPA 에서 지연 로딩과 즉시 로딩은 @OneToOne, @ManyToOne, @OneToMany, @ElementCollection 처럼 관계의 다중성을 표현하는 어노테이션의 fetch 속성으로 정의할 수 있다. fetch 속성의 값을 LAZY 로 지정하면 지연 로딩이 적용되고 EAGER 로 지정하면 즉시 로딩이 적용된다. @OneToOne, @ManyToOne 과 같이 대상 객체의 다중성이 1 인 경우에는 기본 값으로 EAGER 가 설정되어 있고, @OneToMany, @ElementCollection 과 같이 대상 객체의 다중성이 N 인 경우에는 기본 값으로 LAZY 가 설정되어 있다.

Tour 클래스의 매핑 설정을 다시 살펴보자.

```
@Entity
public class Tour {
  @ManyToOne
  @JoinColumn(name="TOURIST_ID")
  private Tourist tourist;

  @ManyToOne
  @JoinColumn(name="TOUR_PACKAGE_ID")
  private TourPackage tourPackage;

  @ElementCollection
  @CollectionTable(name="TOUR_PATH", )
  private List<Path> paths = new ArrayList<>();
}
```

Tourist와 TourPackage는 @ManyToOne으로, Path의 리스트는 @ElementCollection으로 매핑되어 있는 것을 알 수 있다. 따라서 Tour 를 조회하면 fetch 가 EAGER 로 설정된 Tourist 와 TourPackage는 함께 조회되지만 fetch가 LAZY로 설정된 Path의 목록은 실제로 리스트에 접근할 때까지 로딩이 지연된다.

TourPackage 의 매핑을 살펴보면 TourMap 은 @ManyToOne 으로, TourPlanner 는 @OneToOne 으로 매핑되어 있음을 알 수 있다. 앞에서 설명한 것처럼 @ManyToOne 과 @OneToOne 의 fetch 기본값은 EAGER 다. 따라서 JPA 는 TourPackage 를 조회할 때 TourMap 과 TourPlanner 도 함께 조회하도록 쿼리를 생성한다.

```
@Entity
public class TourPackage {
  @ManyToOne
  @JoinColumn(name="TOUR_MAP_ID")
  private TourMap tourMap;

  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name="TOUR_PLANNER_ID")
  private TourPlanner tourPlanner;
}
```

JPA 는 현재의 설정 정보에 따라 Tour 를 조회할 때는 즉시 로딩으로 설정된 Tourist 와 TourPackage 를 함께 로드하고, TourPackage 를 로드하면서 즉시 로딩으로 설정된 TourMap 과 TourPlanner 도 함께 로딩한다. 결과적으로 Tour 를 조회하면 Tourist, TourPackage, TourMap, TourPlanner가 함께 로딩한다.

Tour를 조회한 후 TourMap의 Path 목록 중 하나에 접근하는 코드가 있다고 하자.

```
EntityTransaction transaction = entityManager.getTransaction();
transaction.begin();
```

```
Tour tour = entityManager.find(Tour.class, 1L);
tour.getTourPackage()
        .getTourMap()
        .getPaths()
        .get(0);

transaction.commit();
```

다음은 entityManager.find(Tour.class, 1L)를 실행했을 때 출력되는 SQL 문을 간략하게 표현한 것이다. JPA 구현체로는 Hibernate 를 사용했다. SQL 문을 통해 즉시 로딩으로 설정된 Tourist, TourPackage, TourMap 은 조인을 통해서 한번에 조회한다는 사실을 알 수 있다.

```
// 1) 즉시 로딩(Eager Loading)
select
     ...
from
     tour tour0_
     left outer join tour_package tourpackag1_
       on tour0_.TOUR_PACKAGE_ID=tourpackag1_.id
     left outer join tour_map tourmap2_
       on tourpackag1_.TOUR_MAP_ID=tourmap2_.id
     left outer join tour_planner tourplanne3_
       on tourpackag1_.TOUR_PLANNER_ID=tourplanne3_.id
     left outer join tourist tourist4_
       on tour0_.TOURIST_ID=tourist4_.id
where
     tour0_.id=?
```

하지만 Tour 에서 지연 로딩으로 설정된 Path 는 연관관계를 통해 실제 객체에 접근하기 전까지는 로드하지 않는다. 예제를 실행시키면 tour.getTourPackage().getTourMap().getPaths().get(0)가 호출되는 시점에서야 아래 SQL 문이 실행된다는 사실을 확인할 수 있다.

```
// 2) 지연 로딩(Lazy Loading)
select
     ...
from
     TOUR_MAP_PATH paths0_
where
     paths0_.tour_map_id=?
```

JPQL(Java Persistence Query Language)을 사용할 때는 지연 로딩과 즉시 로딩이 다른 방식으로 적용된다. JPQL 은 SQL 과 유사한 방식의 쿼리를 작성할 수 있는 표준 스펙이지만, 데이터베이스 테이블을 대상으로 하는 SQL 과 달리 객체를 대상으로 쿼리를 작성할 수 있다. JPQL 로 작성된 쿼리를 실행하면 JPA 는 클래스에 설정된 로딩 설정을 무시하고 JPQL 에 선언된 내용만 가지고 쿼리를 실행한다. 일단 객체를 로딩한 후에 클래스의 매핑 정보를 기반으로 필요한 객체들을 추가적으로 로딩한다. 예제를 통해 살펴보자.

다음은 EntityManager 의 find 대신 JPQL 을 사용해서 Tour 를 조회하는 코드를 나타낸 것이다.

```
EntityTransaction entityTransaction = entityManager.getTransaction();
entityTransaction.begin();

Tour tour = entityManager.createQuery(
                  "select t from Tour t where t.id=?1", Tour.class)
                  .setParameter(1, 1L)
                  .getSingleResult();

tour.getTourPackage().getTourMap().getPaths().get(0);

entityTransaction.commit();
```

앞에서 설명한 것처럼 JPQL 을 이용해서 객체를 조회하는 경우에는 로딩 설정을 무시하고 JPQL 만으로 쿼리를 실행한다. 위 코드에서는 기본 키가 1 인 Tour 인스턴스를 조회하는 JPQL 을 실행하기 때문에 Tour 를 조회하는 SQL 문만 실행한다.

```
1) JPQL 실행
select
    ...
from
    tour tour0_
where
    tour0_.id=?
```

JPA 는 Tour 의 조회가 끝난 후에야 Tour 클래스의 매핑 정보를 확인한다. TourPackage 와 Tourist 가 즉시 로딩으로 설정되어 있기 때문에 TourPackage 와 Tourist 를 조회하는 쿼리를 이어서 실행한다. 이 시점에는 원래의 매핑 정보가 그대로 적용되기 때문에 TourPackage 에 즉시 로딩으로 설정된 TourMap 과 TourPlanner 도 함께 로드하게 된다.

```
2) TourPackage 즉시 로딩
select
     ...
from
     tour_package tourpackag0_
     left outer join tour_map tourmap1_
         on tourpackag0_.TOUR_MAP_ID=tourmap1_.id
     left outer join tour_planner tourplanne2_
         on tourpackag0_.TOUR_PLANNER_ID=tourplanne2_.id
where
     tourpackag0_.id=?

3) Tourist 즉시 로딩
select
      ...
from
      tourist tourist0_
```

```
where
      tourist0_.id=?
```

TourMap 이 포함하는 Path 의 목록은 지연 로딩으로 설정되어 있기 때문에 앞의 경우와 마찬가지로 tour.getTourPackage().getTourMap().getPaths().get(0)를 실행하는 시점에 가서야 쿼리가 실행된다.

```
// 4) 지연 로딩(Lazy Loading)
select
      ...
from
     TOUR_MAP_PATH paths0_
where
      paths0_.tour_map_id=?;
```

영속성 전이

지연 로딩과 즉시 로딩이 객체를 로드할 때의 범위를 제어하는데 반해, 객체가 저장될 때의 범위를 제어하는 것을 **영속성 전이(transitive persistence)**라고 부른다. 영속성 전이란 작업 단위에 포함된 객체뿐만 아니라 해당 객체와 연결된 객체들까지 자동으로 영속 대상으로 포함시키는 특성을 말한다.

영속성 전이와 관련해서 알아두면 좋은 개념으로 **도달 가능성에 의한 영속성(persistence by reachability)**이 있다. 이 개념은 어떤 영속 객체로부터 도달 가능한 모든 객체에게 자동으로 영속성이 전이된다는 것을 의미한다. 쉽게 말해서 EntituManager 를 이용해서 Tour 객체를 저장하면 Tour 객체뿐만 아니라 Tour 를 통해 도달 가능한 Tourist, TourPackage, TourMap, TourPlanner 모두가 영속 가능한 상태를 가지게 된다는 것이다.

쉽게 예상할 수 있겠지만 ORM 은 도달 가능성에 의한 영속성을 완벽하게 지원하지는 않는다. 이 특성을 지원하려면 대상 객체로부터 시작하는 모든 연관관계를 추적해서 객체 그래프 전체에 걸쳐 영속성을 적용해야 하기 때문이다. 대신 영속성이 전파되는 범위를 제어할 수 있는 추가적인 매핑 설정을 제공한다. 따라서 작업 단위에 포함된 객체와 연관된 객체라고 해서 무조건 함께 저장되거나 삭제되지는 않는다는 사실에 주의해야 한다.

JPA 에서 영속성 전이는 @OneToOne, @ManyToOne, @OneToMany 와 같은 다중성을 정의하는 어노테이션에 cascade 속성으로 설정할 수 있다. 여행 애플리케이션에서는 TourPackage 를 조작하는 어떤 경우에도 TourPlanner 로 영속성 상태가 전이될 수 있도록 CascadeType.ALL 을 설정했다. 이 설정에 의해 TourPackage 를 저장하거나, 삭제하거나, 수정하면 자동으로 연관된 TourPlanner 역시 함께 저장되거나, 삭제하거나, 수정된다.

```
@Entity
public class TourPackage {
```

```
  @ManyToOne
  @JoinColumn(name="TOUR_MAP_ID")
  private TourMap tourMap;

  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name="TOUR_PLANNER_ID")
  private TourPlanner tourPlanner;
}
```

지금까지 여행 애플리케이션에 적용된 JPA 의 기능과 ORM 의 특성에 대해 간략하게 살펴봤다. 이번 장에서 다루는 내용은 예제를 이해하는데 필요한 범위로 한정돼 있기 때문에 자세한 내용은 앞에서 소개한 자료를 참고하기 바란다.

이제 여행 애플리케이션에 새로운 기능을 추가해 가면서 JPA 를 사용할 때 객체 경계의 관점에서 빠지기 쉬운 함정이 무엇인지 살펴보도록 하자.

기능 추가하기

새로 추가할 기능은 두 가지인데 하나는 시스템이 제공한 여행을 사용자가 확정할 수 있는 기능이고, 다른 하나는 현재 판매중인 상품의 전체 랭킹을 계산하는 기능이다.

여행 확정 기능 추가하기

먼저 TourPackage 에 추가된 인스턴스 변수를 살펴보자. stockCount 는 현재 판매 가능한 상품의 재고를, plannedCount 는 plan 메서드의 결과로 생성된 Tour 인스턴스의 갯수를, confirmedCount 는 이번에 새로 추가하는 기능을 통해 사용자가 최종 확정한 총 갯수를, rank 는 상품의 현재 랭킹을 저장한다.

```
@Entity
public class TourPackage {
  private int rank;
  private int stockCount;
  private int plannedCount;
  private int confirmedCount;
}
```

여행 상품 재고라는 개념을 추가했으므로 이제 재고가 부족할 경우 예외를 던지도록 TourPackage 의 plan 메서드를 수정하자.

```
@Entity
public class TourPackage {
  public Tour plan(Tourist tourist, Location from, Location to) {
    if (stockCount <= 0) {
       throw new CannotPlanTourException(
         "여행 상품 재고가 부족해서 예약을 완료할 수 없습니다.");
```

```
    }

    return tourPlanner.plan(tourist, from, to);
  }

  public void increasePlannedCount() {
    plannedCount++;
  }
}
```

TourPlanner 의 plan 메서드는 Tour 의 인스턴스를 생성하고 TourPackage 의 increasePlannedCount 를 호출해서 plannedCount 의 값을 증가시킨다. Tour 의 생성자에 TourState.PLANNED 라는 인자를 추가로 전달하는데, 이 값은 현재 Tour 의 상태를 계획 중인 상태로 지정하기 위해 전달하는 값이다. 나중에 추가할 확정 기능에서는 다 상태를 CONFIRMED 로 변경할 것이다.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "PLANNER_TYPE")
public abstract class TourPlanner {
  public Tour plan(Tourist tourist, Location from, Location to) {
    ...
    Tour tour = new Tour(tourist, tourPackage, route, TourState.PLANNED);

    validatePlannedTour(tour);

    tourPackage.increasePlannedCount();

    return tour;
  }
}


@Entity
public class TourPackage {
  public void increasePlannedCount() {
    plannedCount++;
  }
}
```

이제 사용자가 여행 계획을 확정하는 기능을 추가하자. 먼저 Tour 클래스에 여행의 상태를 나타내는 열거형 타입 TourState 와 여행의 상태를 저장할 인스턴스 변수 tourState 를 추가하자. @Enumerated 는 열거형 타입을 매핑할 때 쓰는 어노테이션으로 EnumType.STRING 으로 값을 지정하면 열거형 타입 상수의 이름을 그대로 데이터베이스에 저장한다. TourState 의 경우에는 데이터베이스에 "PLANNED"나 "CONFIRMED"라는 문자열이 저장된다.

```
@Entity
public class Tour {
  public enum TourState { PLANNED, CONFIRMED }

  @Enumerated(EnumType.STRING)
```

```
  private TourState tourState;
}
```

사용자가 여행을 확정하면 PLANNED 상태의 Tour 를 CONFIRMED 로 변경해야 한다. 여행의 상태를 변경하는 confirm 메서드를 추가하고, 여행의 상태가 PLANNED 인 경우에만 TOUR 의 상태를 CONFIRMED 로 변경할 수 있으므로 여행의 상태를 체크해야 한다.

```
@Entity
public class Tour {
  public void confirm() {
    if (!tourState.equals(TourState.PLANNED)) {
      throw new CannotConfirmTourException(
        "여행이 계획 상태가 아닙니다. 계획 상태인 여행만 확정할 수 있습니다");
    }

     tourPackage.confirm(this);

    this.tourState = TourState.CONFIRMED;
  }
}
```

TourPackage 의 confirm 메서드는 상품의 재고(stockCount)가 없을 경우에는 예외를 발생시킨다. 재고가 남아 있을 경우 재고(stockCount)와 계획 중인 상품의 수(plannedCount)를 하나씩 감소시키고 확정된 상품의 수(confirmedCount)를 한 개 증가시킨다.

```
@Entity
public class TourPackage {
  public void confirm() {
    if (stockCount <= 0) {
      throw new CannotConfirmTourException(
        "여행 상품 재고가 부족해서 예약을 완료할 수 없습니다.");
    }

    stockCount--;
    plannedCount--;
    confirmedCount++;
  }
}
```

마지막으로 TourService 의 confirm 메서드는 Tour 를 데이터베이스에서 조회한 후 조회한 Tour 의 confirm 메서드를 실행한다.

```
public class TourService {
  public Tour confirm(Long tourId) {
    return unitOfWork.perform(() -> {
       Tour tour = tourDAO.findById(tourId);

       tour.confirm();
```

```
        return tour;
    });
  }
}
```

Tour 의 confirm 메서드가 Tour 의 내부 상태를 변경함에도 불구하고 데이터베이스에 대해 업데이트를 실행하는 구문이 없다는 사실에 주목하라. 이것은 앞에서 설명한 것처럼 ORM 이 데이터베이스로부터 조회한 객체를 **작업 단위(Unit of Work)**에 저장한 후 트랜잭션이 커밋될 때 작업 단위에 저장된 객체의 변경 사항을 추적해서 자동으로 SQL 문을 실행하는 **트랜잭션을 지원하는 쓰기 지연** 기능을 제공하기 때문이다.

랭킹 기능 추가하기

여행 애플리케이션에 추가할 두 번째 기능은 현재 판매중인 상품들을 인기순으로 정렬하고 랭킹을 부여하는 것이다. 시스템은 매일 밤마다 배치를 실행시켜 TourPackage 의 stockCount, plannedCount, confirmedCount 값을 이용해서 랭킹을 계산한 후 인스턴스 변수 rank 에 해당 값을 저장한다.

여기에서는 다음과 같은 공식으로 각 TourPackage 의 점수를 계산한다고 가정한다.

```
score = plannedCount + confirmedCount * 2 – (stockCount – confirmedCount)
```

시스템은 score 의 값이 가장 큰 TourPackage 의 랭킹을 가장 높게 책정한다. 여기에서 랭킹이 가장 높다는 의미는 TourPackage 의 rank 속성에 1 이라는 값이 저장된다는 것을 의미한다. 두 번째 랭킹이 높은 TourPackage 의 rank 속성에는 2 가 저장된다.

먼저 TourPackage 의 점수를 계산하고 저장할 Score 클래스를 살펴보자. Score 클래스는 랭킹을 결정하는데 사용할 값을 저장하는 value 와 이 점수와 연관된 상품을 가리키는 tourPackage 를 인스턴스 변수로 포함한다. calculateScore 메서드는 stockCount, plannedCount, confirmedCount 값을 이용해서 점수를 계산한다. 이렇게 계산된 Score 들을 정렬해서 랭킹을 구할 수 있도록 Comparable 인터페이스를 구현하고 있다.

```
public class Score implements Comparable<Score> {
  private int value;
  private TourPackage tourPackage;

  public Score(TourPackage tourPackage) {
    this.tourPackage = tourPackage;
    this.value = calculateScore(tourPackage);
  }

  private int calculateScore(TourPackage tourPackage) {
    return tourPackage.getPlannedCount()
      + tourPackage.getConfirmedCount() * 2
      - (tourPackage.getStockCount() - tourPackage.getConfirmedCount());
```

```
  }

  public TourPackage getTourPackage() {
    return tourPackage;
  }

  @Override
  public int compareTo(Score other) {
    return other.value - this.value;
  }
}
```

TourPackage 의 점수를 담을 수 있는 Score 클래스를 구현했기 때문에 이제 Score 를 정렬해서 랭킹을 구할 수 있다. 실제 애플리케이션이라면 일정한 청크(chunk) 단위로 랭킹을 처리하겠지만, 여기에서는 뒤에서 살펴볼 데이터 경합 문제를 쉽게 재현하기 위해 전체 TourPackage 를 한번에 조회한 후 Score 를 구하고 정렬하도록 구현했다. RankBatchExecutor 는 Score 의 인스턴스들을 정렬한 후 가장 점수가 높은 TourPackage 부터 차례대로 rank 값을 설정한다.

```
public class RankBatchExecutor {
  private TourPackageDAO tourPackageDAO;
  private UnitOfWork unitOfWork;

  public RankBatchExecutor(TourPackageDAO tourPackageDAO,
      UnitOfWork unitOfWork) {
    this.tourPackageDAO = tourPackageDAO;
    this.unitOfWork = unitOfWork;
  }

  public void rank() {
    unitOfWork.perform(() -> {
      List<TourPackage> tourPackages = tourPackageDAO.findAll();

      List<Score> scores = tourPackages.stream()
                        .map(each -> new Score(each))
                        .sorted()
                        .collect(Collectors.toList());

       for (int current = 0; current < scores.size(); current++) {
         TourPackage tourPackage = scores.get(current).getTourPackage();
         tourPackage.rank(current+1);
       }

       return null;
    });
  }
}
```

모든 TourPackage 를 조회하기 위해 JPQL 로 작성된 쿼리를 사용한다.

```
public class TourPackageDAOImpl extends JpaDAO<TourPackage>
    implements TourPackageDAO {
  @Override
  public List<TourPackage> findAll() {
```

```
    return findMany("select t from TourPackage t", new Object[]{});
  }
}
```

지금까지 사용자가 여행을 확정하는 기능과 여행 상품의 랭킹을 계산하는 기능을 추가했다. 이 예제는 애플리케이션의 기능을 확장하거나 수정할 때 JPA 를 사용해서 얻을 수 있는 이점을 잘 보여준다. 새로운 기능을 추가하면서 데이터베이스의 컬럼을 추가하거나 클래스의 인스턴스 변수를 추가했음에도 SQL 문을 추가하거나 수정할 필요가 없었다. 여행을 계획하는 과정에서 TourPackage 의 plannedCount 값을 증가시키는 작업이 추가됐지만 데이터베이스에 수정사항을 반영할 코드를 추가로 작성할 필요는 없었다. TourService 의 plan 메서드를 보면 plannedCount 를 추가하기 전과 후에 차이가 없다는 사실을 알 수 있다. 이것은 TourPackage 의 상태 수정을 JPA 가 자동으로 인식하고 업데이트 쿼리를 생성하기 때문이다.

```
public class TourService {
  public Tour plan(Long touristId, Long tourPackageId,
      Location from, Location to) {
    return unitOfWork.perform(() -> {
      Tourist tourist = touristDAO.findById(touristId);
      TourPackage tourPackage = tourPackageDAO.findById(tourPackageId);

      Tour tour = tourPackage.plan(tourist, from, to);
      tourDAO.save(tour);

      return tour;
    });
  }
}
```

JPA 의 지원이 없었다면 다음과 같이 Tour 를 저장한 후에 명시적으로 TourPackage 를 업데이트하는 코드를 추가해야 했을 것이다.

```
public class TourService {
  public Tour plan(Long touristId, Long tourPackageId,
      Location from, Location to) {
    return unitOfWork.perform(() -> {
      Tourist tourist = touristDAO.findById(touristId);
      TourPackage tourPackage = tourPackageDAO.findById(tourPackageId);

      Tour tour = tourPackage.plan(tourist, from, to);

      tourDAO.save(tour);
      tourDAO.modify(tour); // JPA 의 경우 명시적으로 업데이트할 필요가 없다

      return tour;
    });
  }
}
```

새로운 기능을 위해 새로운 객체에게 메시지를 전송해야 하더라도 연관관계만 설정되어 있다면 원하는 객체에 쉽게 도달할 수 있다. 결과적으로 JPA 는 객체 사이의 경계 문제를 간단하게 해결해준다. 연관관계를 통해 다른 객체에 접근할 때 JPA 는 클래스에 설정된 지연 로딩과 즉시 로딩 매핑을 이해하고 적절한 시점에 필요한 객체들을 로딩한다. 따라서 하나의 트랜잭션 안에서 조회할 객체의 경계를 미리 결정해야하는 필요성을 없애준다. 원하는 객체를 로딩하고 연관관계를 따라 이동하거나 메시지를 전송하면 JPA 가 알아서 원하는 객체를 조회하고 수정된 사항을 반영해 주기 때문이다.

그러나 객체의 경계 문제를 피하기 위해 무비판적으로 JPA 가 제공하는 특성을 받아들이면 애플리케이션에 심각한 문제가 발생할 수 있다. 사람들은 종종 객체의 경계에 대해 고민하는 것은 도메인 관심사를 코드에 올바르게 담기 위해 가장 중요한 요소라는 사실을 간과한다. 결과적으로 JPA 를 객체지향 설계를 위한 만능 해결책으로 생각하는 순간 다양한 문제를 떠안는 동시에 가치 있는 도메인 제약사항을 고민할 수 있는 기회도 잃어버리게 되고 만다.

관심사 분리의 함정

JPA 는 데이터베이스 처리와 관련된 다양한 이슈를 감추고 객체를 투명하게 처리할 수 있는 다양한 기능을 제공한다. 어떤 사람들은 JPA 를 전통적인 객체지향 애플리케이션을 구현하기 위한 필수품으로 생각하기도 한다. 하지만 JPA 에 대한 이해가 부족한 상황에서 섣부르게 적용할 경우 데이터베이스의 매핑과 성능 이슈를 해결하기 위한 코드에 도메인 로직이 압도당하게 된다.

잊지 말아야 할 사실은 애플리케이션의 성공을 결정하는 것은 기술적인 요소가 아니라 도메인 로직이며 도메인 관심사가 기술 관심사에 침식되지 않도록 명확하게 관심사를 분리해야 한다는 점이다. 관심사를 모듈이나 레이어로 나누고 관심사 사이의 의존성 방향을 주의 깊게 설계하는 이유는 한번에 하나의 관심사에만 집중하고 덜 중요한 관심사를 다루는 코드가 변경되더라도 더 중요한 관심사를 다루는 코드에 영향이 없도록 하기 위해서다.

관심사 분리가 잘 된 애플리케이션에서는 도메인 관심사에 대해 고민할 때 기술적인 요소에 대해서는 고민하지 않아도 무방할 것이다. 여기에서 핵심은 '고민하지 않아도 된다'는 점이다. 앞에서 살펴본 것처럼 도메인 관심사와 기술 관심사가 한데 뒤섞이면 애플리케이션의 초점이 흐려지고 코드를 수정하고 유지보수하기 어려워진다. 하지만 중요한 것은 코드가 분리되어 있는지 여부가 아니다. 도메인 로직에 관해 고민할 때 기술적인 요소에 관해 고민하지 않을 수 있는 자유가 있는지가 핵심이다. 코드가 분리되어 있더라도 기술적인 관심사에 대해 '고민'해야 한다면 관심사가 제대로 분리되지 않은 것이다.

JPA 가 제공하는 다양한 기능과 ORM 특유의 복잡성은 개발자들을 쉽게 함정에 빠져들게 유인한다. 도메인 객체에서 데이터베이스 처리 로직을 덜어냈기 때문에 데이터베이스와 관련된 다양한 기술 관심사를 도메인 레이어로부터 제거했다고 착각하게 된다. 하지만 실제로 JPA 를 원활하게 사용하기 위해서는 도메인 로직에 관해 '고민'하는 동시에 데이터베이스에 관해

'고민'해야만 한다. 다시 말해서 코드 레벨에서는 도메인 관심사와 데이터베이스 관심사를 분리하지만 사고의 관점에서는 이 둘을 뒤섞어 놓는다는 것이다. 이제부터 이유를 살펴보자.

객체 조회

TourService 의 confirm 메서드는 데이터베이스로부터 Tour 인스턴스를 조회한 후 여행을 확정하기 위해 조회된 객체에게 confirm 메시지를 전송한다. 그리고 Tour 객체는 다시 TourPackage 의 confirm 메시지를 호출해서 확정된 상품의 개수를 증가시킨다.

Tour 와 TourPackage 의 JPA 설정을 살펴보면 Tour 를 조회할 때마다 즉시 로딩이 설정된 Tourist, TourPackage, TourMap, TourPlanner 가 함께 조회된다는 사실을 알 수 있다.

```
@Entity
public class Tour {
  @ManyToOne
  @JoinColumn(name="TOURIST_ID")
  private Tourist tourist;

  @ManyToOne
  @JoinColumn(name="TOUR_PACKAGE_ID")
  private TourPackage tourPackage;
}

@Entity
public class TourPackage {
  @ManyToOne
  @JoinColumn(name="TOUR_MAP_ID")
  private TourMap tourMap;

  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name="TOUR_PLANNER_ID")
  private TourPlanner tourPlanner;
}
```

따라서 TourService 의 confirm 메서드를 호출하면 JPA 는 아래와 같이 Tour, Tourist, TourPackage, TourMap, TourPlanner 를 조인으로 연결하는 SQL 문을 실행할 것이다.

```
select
     ...
from
     tour tour0_ left outer join tour_package tourpackag1_
       on tour0_.TOUR_PACKAGE_ID=tourpackag1_.id
     left outer join tour_map tourmap2_
      on tourpackag1_.TOUR_MAP_ID=tourmap2_.id
     left outer join tour_planner tourplanne3_
      on tourpackag1_.TOUR_PLANNER_ID=tourplanne3_.id
     left outer join tourist tourist4_
       on tour0_.TOURIST_ID=tourist4_.id
where
     tour0_.id=?
```

TourService 에서 호출된 Tour 는 다시 TourPackage 의 confirm 메서드를 호출해서 상품의 판매 개수를 증가시킨다.

```
@Entity
public class TourPackage {
  public void confirm() {
    if (stockCount <= 0) {
      throw new CannotConfirmTourException(
        "여행 상품 재고가 부족해서 예약을 완료할 수 없습니다.");
    }

    stockCount--;
    plannedCount--;
    confirmedCount++;
  }
}
```

여기에서 한 가지 마음에 걸리는 부분은 상품을 확정할 때 Tour 와 TourPackage 는 반드시 필요하지만 Tourist 와 TourMap, TourPlanner 는 전혀 필요하지 않다는 점이다. 하지만 현재의 매핑 설정에 따르면 Tour 를 조회할 때마다 TourPackage 뿐만 아니라 Tourist, TourMap, TourPlanner 까지 함께 조회된다. 기능과는 아무런 상관이 없는 불필요한 데이터도 함께 조회되는 것이다.

랭킹을 계산할 때는 TourPackage 의 속성을 이용해서 점수를 계산하고, 계산된 결과를 이용해서 TourPackage 의 rank 속성을 변경한다. 따라서, 랭킹을 계산할 때는 TourPackage 만 조회하면 된다. 하지만 TourPackage 에 선언된 즉시 로딩 설정으로 인해 TourMap 과 TouePlanner 도 항상 함께 조회된다. 이 경우에도 불필요한 데이터가 조회되는 것이다.

랭킹 계산을 위한 쿼리는 JPQL 로 작성되어 있다. JPA 는 JPQL 구문에 나타난 객체만을 이용해서 SQL 을 실행한 후에 로드된 객체에 설정된 매핑 설정을 기반으로 추가적인 SQL 문을 실행한다는 점을 기억하라. 따라서 랭킹 계산 시에 JPA 는 TourPackage 의 목록을 조회한 후 즉시 로딩으로 설정된 TourMap, TourPlanner 를 별도의 쿼리를 이용해서 이어서 조회한다.

```
1) JPQL "select t from TourPackage t" 조회
select
     ...
from
     tour_package tourpackag0_

2) TourPackage -> TourMap @ManyToOne 즉시로딩 매핑
select
     ...
from
     tour_map tourmap0_
where
     tourmap0_.id=?
```

```
3) TourPackage -> TourPlanner @OneToOne 즉시로딩 매핑
select
     ...
from
     tour_planner tourplanne0_
     left outer join
     tour_package tourpackag1_
     on tourplanne0_.id=tourpackag1_.TOUR_PLANNER_ID
     left outer join
     tour_map tourmap2_
     on tourpackag1_.TOUR_MAP_ID=tourmap2_.id
where
     tourplanne0_.id=?

3) TouePlanner -> TourPackage @OneToOne 즉시로딩 매핑
select
     ...
from
     tour_package tourpackag0_
     left outer join tour_map tourmap1_
       on tourpackag0_.TOUR_MAP_ID=tourmap1_.id
     left outer join tour_planner tourplanne2_
       on tourpackag0_.TOUR_PLANNER_ID=tourplanne2_.id
where
     tourpackag0_.TOUR_PLANNER_ID=?
```

한 가지 눈 여겨 볼 부분은 TourPackage 와 TourPlanner 사이에 설정된 양방향 연관관계를 동기화시키기 위해 두 객체를 조회하는 쿼리가 여러 번 실행된다는 점이다. 이 예제는 성능 측면에서도 양방향 연관관계를 피해야 한다는 사실을 잘 보여준다.

이 문제를 해결할 수 있는 한 가지 방법은 Tour 와 TourPackage 만 조회되도록 Tourist 와 TourMap, TourPlanner 에 대한 fetch 속성을 지연 로딩으로 변경하는 것이다.

```
@Entity
public class Tour {
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name="TOURIST_ID")
  private Tourist tourist;

  @ManyToOne
  @JoinColumn(name="TOUR_PACKAGE_ID")
  private TourPackage tourPackage;
}

@Entity
public class TourPackage {
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name="TOUR_MAP_ID")
  private TourMap tourMap;

  @OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
  @JoinColumn(name="TOUR_PLANNER_ID")
  private TourPlanner tourPlanner;
}
```

이제 TourService 의 confirm 메서드를 호출하면 다음과 같이 Tour 와 TourPackage 만 조인에 포함시키고 Tourist 와 TourMap, TourPackage 는 조인에서 제외된다는 것을 알 수 있다.

```
select
     ...
from
    tour tour0_ left outer join tour_package tourpackag1_
      on tour0_.TOUR_PACKAGE_ID=tourpackag1_.id
where
    tour0_.id=?
```

상품의 랭킹을 계산하는 서비스에서 TourPackage 를 조회하는 SQL 역시 변경 후에는 TOUR_PACKAGE 테이블만 쿼리한다.

```
select
     ...
from
     tour_package tourpackag0_
```

문제는 클래스에 선언된 로딩은 전역 설정이라는 것이다. 따라서 TourPackage 에 선언된 즉시로딩을 지연로딩으로 변경하면 기존의 plan 메서드의 로딩 방식도 지연 로딩으로 변경된다.

지연 로딩으로 변경하기 전의 plan 메서드는 다음과 같이 TourPackage 와 TourPlanner, TourMap 을 하나의 쿼리로 조회할 수 있었다.

```
select
     ...
from
     tour tour0_ left outer join tour_package tourpackag1_
       on tour0_.TOUR_PACKAGE_ID=tourpackag1_.id
     left outer join tour_map tourmap2_
       on tourpackag1_.TOUR_MAP_ID=tourmap2_.id
     left outer join tour_planner tourplanne3_
       on tourpackag1_.TOUR_PLANNER_ID=tourplanne3_.id
     left outer join tourist tourist4_
       on tour0_.TOURIST_ID=tourist4_.id
where
     tour0_.id=?
```

그러나 변경한 후의 plan 메서드는 지연 로딩 설정으로 인해 TourPackage, TourPlanner, TourMap 을 조회하기 위한 쿼리를 독립적으로 실행한다.

```
1) TourPackage만 로딩
select
     ...
from
```

```
      tour_package tourpackag0_
where
      tourpackag0_.id=?
```

2) TourPlanner 지연로딩

```
select
      ...
from
      tour_planner tourplanne0_left outer join tour_package tourpackag1_
        on tourplanne0_.id=tourpackag1_.TOUR_PLANNER_ID
where
      tourplanne0_.id=?
```

3) TourMap 지연로딩

```
select
      ...
from
      tour_map tourmap0_
where
      tourmap0_.id=?
```

정리해보자. 여행을 확정하거나 랭킹을 계산하는 기능에서는 불필요한 객체 조회를 방지할 수 있기 때문에 지연로딩으로 설정하는 편이 유리하다. 반면에 여행을 계획하는 기능에서는 필요한 객체들을 한번에 죄회할 수 있는 즉시로딩이 유리하다. 이런 상황에 맞닥뜨린 대부분의 개발자는 더 많은 기능에 적용할 수 있는 설정을 선택한다. 예제의 경우 여행을 계획하는 기능보다는 여행을 확정하고 랭킹을 계산하는 기능이 더 빈번하게 사용되기 때문에 지연로딩을 기본 매핑으로 설정할 수 있다. 대신 여행을 계획하는 기능에서는 JPQL 의 페치 로딩(fetch loading)을 이용해서 일시적으로 TourMap 과 TourPlanner 를 함께 조회하도록 지정할 수 있다.

```
public class TourPackageDAOImpl extends JpaDAO<TourPackage>
    implements TourPackageDAO {
  @Override
  public TourPackage findForPlanning(Long id) {
    return findOne("select p " +
                   "from TourPackage p " +
                   "     left join fetch p.tourPlanner " +
                   "     left join fetch p.tourMap " +
                   "where p.id=?1",
                   new Object[] {id});
  }
}
```

JPA 는 조회되는 객체의 범위를 유연하게 지정할 수 있는 다양한 방법을 지원한다. 보편적으로 사용될 전역 설정은 클래스에 설정하지만 성능 튜닝이 필요하거나 쿼리가 복잡한 경우에는 전역 설정을 무시하고 JPQL 을 사용해서 조회 범위를 조종할 수 있다. 따라서 클래스에 설정된 매핑 정보는 단지 힌트일 뿐이고 실제로 조회되는 객체의 범위는 조회를 수행하는 코드를 살펴본 후에야 알 수 있게 된다. 새로운 기능을 추가하는 개발자는 쿼리를 최적화하기 위해 새로운 JPQL 을 추가할 것이고, 실행되는 쿼리의 종류는 점점 더 늘어날 것이다. 결과적으로 애플리케이션 안에는 객체 경계에 대한 기준을 찾아볼 수 없게 된다.

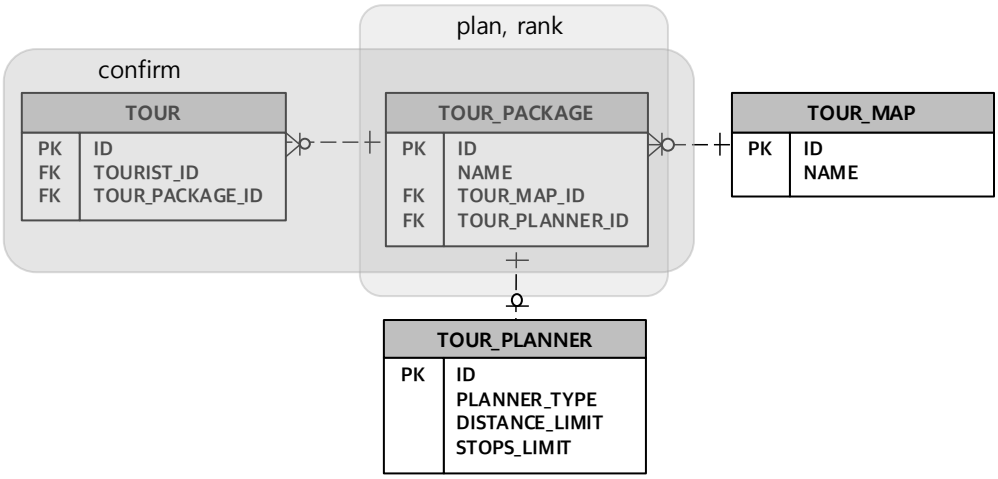
나중에 자세히 다루겠지만 객체의 범위는 도메인 제약을 기준으로 정해야 한다. 도메인 제약에 초점을 맞추면 전체적으로 일관성 있는 객체 경계를 결정할 수 있다. 성능은 중요하다. 하지만 도메인 규칙과 제약이 더 중요하다. 도메인 규칙과 제약을 만족시키는 객체 경계를 정한 후 이 경계에 따라 성능을 최적화해야 한다.

여기에서 말하고자 하는 핵심은 JPA 가 제공하는 유연성이 조회되는 객체의 범위에 대해 충분히 고민할 필요성을 느끼지 못하게 만든 것이다. 이것은 JPA 를 사용하는 경우에 발생하는 가장 최악의 결과인데 본래 의도와 다르게 도메인이 아니라 데이터베이스 성능에 집중하게 만들기 때문이다.

트랜잭션 경계

이상하게 들릴 수도 있겠지만 도메인 관점에서 객체의 경계를 고려하지 않을 경우 데이터베이스 측면에서 트랜잭션 경합이 발생할 가능성도 높아진다. 현재 구현된 기능에서 트랜잭션 안에서 수정되는 객체 경계를 살펴보자. TourService 의 plan 메서드는 Tour 를 생성하고 TourPackage 의 상태를 수정한다.

따라서 TOUR 테이블에 인서트 쿼리가 실행되고 TOUR_PACKAGE 테이블에는 업데이트가 실행된다. TourService 의 confirm 메서드는 Tour 와 TourPackage 의 상태를 수정한다. 따라서 TOUR 테이블과 TOUR_PACKAGE 테이블이 업데이트된다. RankBatchExecutor 의 rank 메서드는 TourPackage 의 상태를 수정한다. 따라서 TOUR_PACKAGE 테이블을 업데이트한다.



그림과 같이 confirm 메서드의 트랜잭션 경계는 TOUR 와 TOUR_PACKAGE 에 걸쳐 있고, plan 과 rank 의 트랜잭션 경계는 TOUR_PACKAGE 만 포함된다. 또한 테이블의 수정 순서나 규칙이 정해져 있지 않고 기능 별로 독립적으로 구현되고 있기 때문에 데이터 경합(data contention) 문제가 발생할 수 있다. plan, confirm, rank 메서드는 모두 TourPackage 를 잠글 수 있기 때문에 이 기능 중 하나라도 트랜잭션이 길어지면 다른 트랜잭션들이 대기 상태에 빠지게 되고 결과적으로 타임아웃 에러가 발생하게 된다.

고작 한두 개 테이블을 수정하는 작업때문에 오류가 발생할 지 의심스러울 수도 있다. 설명을 위해 문제를 과도하게 부풀린 거라고 생각할 지도 모른다. 물론 데이터량도 많지 않고 유입되는 트래픽도 적다면 고작 한두 개의 테이블을 업데이트하는 작업만으로는 이런 문제가 발생하지는 않는다. 하지만 데이터와 트래픽은 서서히 증가하지 않는다. 데이터량도, 트래픽도 서비스가 널리 알려지는 순간 급작스럽게 증가하게 된다. 그리고 애플리케이션의 코드나 인프라 구성 등은 큰 폭으로 증가한 데이터와 트래픽을 감당할 수 있는 준비가 되지 못한 상태인 경우가 대부분이다. 그 결과 사용자가 원활하게 서비스를 이용할 수 없을 만큼 빈번하게 장애가 발생하게 된다.

예제 애플리케이션을 실행해 보면 랭킹 배치가 TOUR_PACKAGE 의 모든 레코드를 잠그기 때문에 여행을 계획하거나 확정하는 시점에 에러가 발생한다는 사실을 확인할 수 있다.

```
Jul 21, 2021 1:46:26 PM org.hibernate.engine.jdbc.spi.SqlExceptionHelper
logExceptions
WARN: SQL Error: 1205, SQLState: 40001
Jul 21, 2021 1:46:26 PM org.hibernate.engine.jdbc.spi.SqlExceptionHelper
logExceptions
ERROR: Lock wait timeout exceeded; try restarting transaction
Jul 21, 2021 1:46:26 PM
org.hibernate.internal.ExceptionMapperStandardImpl mapManagedFlushFailure
ERROR: HHH000346: Error during managed flush [could not execute
statement]
```

예제에서 에러가 발생한 원인은 무엇일까? 랭킹 배치를 보면 하나의 트랜잭션 안에서 TOUR_PACKAGE 의 모든 레코드를 읽은 후 업데이트하고 있다.

```
public class RankBatchExecutor {
  public void rank() {
    unitOfWork.perform(() -> {
      // 하나의 트랜잭션 안에서 전체 TourPackage 를 로드한 후
      List<TourPackage> tourPackages = tourPackageDAO.findAll();
      ...
      // 전체 TourPackage 를 업데이트 한다
      for (int current = 0; current < scores.size(); current++) {
        TourPackage tourPackage = scores.get(current).getTourPackage();
        tourPackage.rank(current+1);
      }
      ...
    });
  }
}
```

배치를 구현했던 개발자는 TOUR_PACKAGE 테이블에 저장된 상품의 수가 많지 않기 때문에 전체 데이터를 한번에 업데이트해도 무방했다. 서비스가 사람들 사이에 알려지고 상품이 늘어나면서 트랜잭션 시간이 조금씩 늘어났지만 문제가 발생하지는 않았다. 그렇게 랭킹 배치는 개발자들에게 서서히 잊혀져 갔다. 시간이 지나고 또 다른 개발자가 여행을 확정하는 기능을 개발하면서 confirm 메서드를 추가했다. confirm 메서드 역시 TOUR_PACKAGE 테이블을 업데이트하지만 여전히 데이터도 적고 트래픽도 낮기 때문에 아무런 문제도 발생하지 않는다.

하지만 성공하는 서비스의 데이터와 트래픽은 순차적으로 증가하지 않고 갑작스럽게, 그것도 예상하지 못했던 수준으로 증가한다. 그리고 어제까지 아무런 문제 없이 잘 실행되던 코드가 어느 순간부터 에러를 뱉어내기 시작한다. 개발팀은 장애의 원인을 찾기 위해 분주히 뛰어다니고, 사용자 C/S 는 폭주하고, 사업 부서는 언제 장애가 해결될 수 있을지 알지 못한 채 발만 동동 구르고 있다. 지어낸 이야기라고 생각할 수도 있지만 실제로 서비스를 운영하다 보면 서비스가 성장하면서 대량의 데이터를 업데이트하는 배치로 인해 사용자 기능에 장애가 발생하는 상황에 자주 마주하게 된다.

원인은 개발자들 사이에 객체가 수정되는 범위와 수정 빈도에 대해서 공감대가 형성되거나 공유가 되지 않았기 때문이다. 아마 사용자가 많지 않던 시절에는 이런 이슈에 대해 이야기할 이유조차 없었을지도 모른다. 수정 가능한 객체 범위에 대한 어떤 제약도 없기 때문에 또 다른 개발자는 또 다른 기능을 추가하면서 더 많은 테이블을 업데이트하도록 코드를 구현하고 있을지도 모른다. 예제에서는 이해를 돕기 위해 단지 두 개의 테이블을 수정하는 예를 소개했지만, 실제로 운영되고 있는 코드에서는 공유된 규칙 없이 무수히 많은 테이블들을 업데이트하는 기능들이 동시에 실행되고 있을 것이다.

기능이 추가될 수록 개발자들은 현재의 요구사항에 적합한 객체들을 조회하고 연관관계를 따라 탐색하면서 객체들의 상태를 수정할 것이다. 그리고 어떤 기능들이 해당 객체를 수정하기 위해 경합하는지, 어떤 빈도로 수정되는지, 수정 사항을 데이터베이스에 반영하기 위해 트랜잭션이 잠금을 점유하는 시간 등에 대한 고려 없이 각자 원하는 기능에 맞게 수정할 객체 범위를 선택할 것이다. 그리고 데이터와 트래픽이 늘어난 순간 조용하던 시스템은 순식간에 붕괴되고 만다.

도메인 로직을 설계할 때는 항상 함께 수정될 객체의 경계와 빈도를 고민해야 한다. 그리고 개발자들은 이 경계를 인식하고 전체 기능에 걸쳐 일관성 있게 적용해야 한다. 사용하는 언어가 무엇이고, 어떤 기술을 사용하고, 하위의 영속성 메커니즘이 무엇인지는 그렇게 중요하지 않다.

JPA 를 사용할 때는 상황이 더 악화되는 경우가 많은데 객체의 연관관계를 통해 어떤 객체라도 접근할 수 있기 때문에 객체 경계에 대해 인식하지 못하게 된다는 점이다. 어떤 객체를 조회하더라도 다른 객체로 이동할 수 있는 참조만 존재한다면 그 참조를 따라 어디라도 이동할 수 있다. 이동하던 중에 거치게 되는 어떤 객체라도 메서드를 호출해서 객체의 상태를 수정할 수 있으며 트랜잭션이 종료되는 시점에 수정된 객체들의 변경 사항이 한데 모여 데이터베이스에 업데이트될 것이다. 지연 로딩과 영속성 전이와 같은 JPA 기능은 객체 참조를 통해 어떤 객체라도 도달할 수 있도록 해주기 때문에 개발자들은 참조를 통한 이동을 시작할 객체와 이동을 종료할 객체에 대해 크게 고민하지 만든다.

JPA 를 사용해서는 안된다는 말이 아니다. JPA 를 사용하면 도메인 관심사를 코드로 명확하게 표현하면서도 데이터베이스 이슈를 투명하고 직관적으로 해결할 수 있다. 객체의 경계를 고려하지 않으면 어떤 기술을 사용하더라도 앞에서 설명한 문제는 언제라도 발생할 수 있다. 여기에서 이야기하고 싶은 이야기는 JPA 가 다른 기술에 비해 객체 참조를 통한 이동을 용이하게 만들기 때문에 상대적으로 이런 문제가 발생하기 더 쉬워진다는 것이다.

이번 장을 요약하면 트랜잭션 안에서 수정되어야 하는 객체의 경계를 결정하고 그 경계를 코드 안에서 명확하게 표현해야 한다는 것이다. 경계에 대한 고민 없이 어떤 객체라도 접근할 수 있는 ORM 의 기능을 애플리케이션에 적용하면 불필요한 데이터 조회와 데이터 경합 문제가 직면하게 된다. 경계에 관해 고민하고 경계에 관해 고민하라. 기술 관점이 아니라 도메인 관점을 기준으로 경계를 결정하라. JPA 와 같은 특정 기술은 문제를 해결해 줄 수 없다. 기술이 문제 해결에 도움은 되겠지만 객체의 경계나 수정 빈도에 대한 고민 없이 맹목적으로 기술에 의존하면 더 심각한 문제와 마주쳐야할 수 있다.

지금까지 몇 장에 걸쳐 관심사를 레이어로 분리하고 레이어 사이의 의존성을 결정하는 방식을 살펴봤다. 레이어 분할 방식은 다양할 수 있지만 한 가지 중요한 원칙은 모든 의존성은 도메인 레이어를 향해야 한다는 것이다. 다시 한번 강조하지만 도메인 레이어의 코드에는 기술적인 관심사가 섞여서는 안된다. 그러나 코드가 섞이지 않는 것만으로는 부족하다. 이번 장에서 살펴본 것처럼 도메인 레이어를 구현할 때 데이터베이스와 같은 기술적인 관심사에 대한 고민이 도메인 레이어의 설계를 주도해서는 안된다. 반대로 도메인 레이어가 데이터베이스와 같은 인프라스트럭처 이슈를 주도해야 한다. 이어지는 장에서는 객체지향 관점에서 도메인 레이어를 설계하는 방법에 관해 살펴보기로 하자.