- **엔티티 매핑**
  - **@Entity**
  - **@Table**
  - **다양한 매핑 사용해보기**
  - **데이터베이스 스키마 자동 생성하기**
  - **DDL 생성 기능**
  - **기본 키(Primary Key) 매핑**
    - - 기본 키 직접 할당 전략
    - - IDENTITY 전략
    - - SEQUENCE 전략
      - – @SequenceGenerator
    - - TABLE 전략
      - – @TableGenerator
    - - AUTO 전략
    - - [참고] 권장하는 식별자 선택 전략
  - **필드와 컬럼 매핑 - 레퍼런스**
    - - @Column
    - - @Enumerated
    - - @Temporal
    - - @Lob
    - - @Transient
    - - @Access

- **[실전 예제] - 1. 요구사항 분석과 기본 매핑**
  - **요구사항 분석**
  - **도메인 모델 분석**
  - **테이블 설계**
  - **엔티티 설계와 매핑**
  - **데이터 중심 설계의 문제점**

엔티티 매핑

JPA를 사용하는데 있어 가장 중요한 일은 엔티티와 테이블을 정확히 매핑하는 것이다. 따라서 매핑 어노테이션을 숙지하고 사용해야 한다.

JPA는 다양한 매핑 어노테이션을 지원하는데 크게 4가지로 분류할 수 있다. 오른쪽에는 대표 어노테이션들을 적어보았다.

- 객체와 테이블 매핑: `@Entity`, `@Table`
- 기본 키 매핑: `@Id`
- 필드와 컬럼 매핑: `@Column`
- 연관관계 매핑: `@ManyToOne`, `@JoinColumn`

이번 장에서는 객체와 테이블 매핑, 기본 키 매핑, 필드와 컬럼 매핑에 대해 알아보고 연관관계 매핑은 5, 6, 7, 8장에 걸쳐서 설명하겠다.

필드와 컬럼을 매핑하는 어노테이션은 기능을 하나하나 설명하기에는 내용이 많으므로 필요할 때 찾아볼 수 있도록 마지막에 레퍼런스 형식으로 정리해두었다.

먼저 객체와 테이블 매핑 어노테이션부터 알아보자.

> **참고:** 매핑 정보는 XML이나 어노테이션 중에 선택해서 기술하면 되는데 책에서는 어노테이션만 사용하겠다. 각각 장단점이 있지만 어노테이션을 사용하는 쪽이 좀 더 쉽고 직관적이다. XML을 사용해서 매핑 정보를 구성하는 방법은 설명하지 않는다.

@Entity

JPA를 사용해서 테이블과 매핑할 클래스는 `@Entity` 어노테이션을 필수로 붙어야 한다. `@Entity`가 붙은 클래스는 JPA가 관리하는 것으로, 엔티티라 부른다.

속성 정리

속성	기능	기본값
name	JPA에서 사용할 엔티티 이름을 지정한다. 보통 기본값인 클래스 이름을 사용한다. 만약 다른 패키지에 이름이 같은 엔티티 클래스가 있다면 이름을 지정해서 충돌하지 않도록 해야 한다.	설정하지 않으면 클래스 이름을 그대로 사용한다. (예: Member)

`@Entity` 적용시 주의사항

- 기본 생성자는 필수다. (파라미터가 없는 `public` 또는 `protected` 생성자)
- `final` 클래스, `enum`, `interface`, `inner` 클래스에는 사용할 수 없다.
- 저장할 필드에 `final`을 사용하면 안 된다.

JPA가 엔티티 객체를 생성할 때 기본 생성자를 사용하므로 이 생성자는 반드시 있어야 한다. 자바는 생성자가 하나도 없으면 다음과 같은 기본 생성자를 자동으로 만들어 준다.

```
public Member(){} //기본 생성자
```

문제는 다음과 같이 생성자를 하나 이상 만들면 자바는 기본 생성자를 자동으로 만들지 않는다. 이때는 기본 생성자를 직접 만들어야 한다.

```
public Member(){} //직접 만든 기본 생성자

//임의의 생성자
public Member(String name){
    this.name = name
}
```

@Table

`@Table`은 엔티티와 매핑할 테이블을 지정한다. 생략하면 매핑한 엔티티 이름을 테이블 이름으로 사용한다.

속성 정리

속성	기능	기본값
name	매핑할 테이블 이름	엔티티 이름을 사용한다.
catalog	catalog 기능이 있는 데이터베이스에서 catalog 를 매핑	
schema	schema 기능이 있는 데이터베이스에서 schema 를 매핑	
uniqueConstraints(DDL)	DDL 생성 시에 유니크 제약조건을 만든다. 2개 이상의 복합 유니크 제약조건도 만들 수 있다. 참고로 이 기능은 스키마 자동 생성 기능을 사용해서 DDL을 만들 때만 사용된다.	

DDL 생성기능은 조금 뒤에 알아보겠다.

다양한 매핑 사용해보기

> **예제 코드 : `ch04-jpa-start2`**

JPA 시작하기 장에서 개발하던 회원 관리 프로그램에 다음 요구사항이 추가되었다.

1. 회원은 일반 회원과 관리자로 구분해야 한다.
2. 회원 가입일과 수정일이 있어야 한다.
3. 회원을 설명할 수 있는 필드가 있어야한다. 이 필드는 길이 제한이 없다.

요구사항을 만족하도록 회원 엔티티에 기능을 추가하자.

```java
package jpabook.start;

import javax.persistence.*;  //**
import java.util.Date;

@Entity
@Table(name="MEMBER")
public class Member {

    @Id
    @Column(name = "ID")
    private String id;

    @Column(name = "NAME")
    private String username;

    private Integer age;

    //=== 추가 ===
    @Enumerated(EnumType.STRING)
    private RoleType roleType; //<1>

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdDate;  //<2>

    @Temporal(TemporalType.TIMESTAMP)
    private Date lastModifiedDate; //<2>

    @Lob
    private String description; //<3>

    //Getter, Setter
    ...
}
```

```java
package jpabook.start;

public enum RoleType {
    ADMIN, USER
}
```

<1> `roleType` : 자바의 `enum` 을 사용해서 회원의 타입을 구분했다. 일반 회원은 `USER` , 관리자는 `ADMIN` 이다. 이처럼 자바의 `enum` 을 사용하려면 `@Enumerated` 어노테이션으로 매핑해야 한다. 속성 값에 대한 자세한 내용은 뒤에 나오는 `@Enumerated` 를 참고하자.

<2> `createdDate`, `lastModifiedDate` : 자바의 날짜 타입은 `@Temporal`을 사용해서 매핑한다. 속성 값에 대한 자세한 내용은 뒤에 나오는 `@Temporal`을 참고하자.

<3> `description` : 회원을 설명할 수 있는 필드에는 길이 제한이 없어야 하므로 데이터베이스의 VARCHAR 타입 대신에 CLOB 타입으로 저장해야 한다. `@Lob`을 사용하면 `CLOB`, `BLOB` 타입을 매핑할 수 있다. 자세한 내용은 뒤에 나오는 `@Lob`를 참고하자.

지금까지는 테이블을 먼저 생성하고 그 다음에 엔티티를 만들었지만, 이번에는 데이터베이스 스키마 자동 생성하기를 사용해서 엔티티만 만들고 테이블은 자동 생성되도록 해보자.

데이터베이스 스키마 자동 생성하기

JPA는 매핑정보를 활용해서 데이터베이스의 스키마를 자동으로 생성하는 기능을 지원한다. 클래스의 매핑정보를 보면 어떤 테이블에 어떤 컬럼을 사용하는지 알 수 있다. JPA는 이 매핑정보와 데이터베이스 방언을 사용해서 데이터베이스 스키마를 생성한다. 참고로 방언은 JPA 시작하기 장에서 설명했다.

스키마 자동 생성 기능을 사용해보자.

먼저 `persistence.xml`에 다음 속성을 추가하자.

```
<property name="hibernate.hbm2ddl.auto" value="create" />
```

이 속성을 추가하면 **애플리케이션 실행 시점에 데이터베이스 테이블을 자동으로 생성**한다. 참고로 `hibernate.show_sql` 속성을 `true`로 설정하면 콘솔에 실행되는 테이블 생성 DDL(Data definition language)을 출력할 수 있다.

```
<property name="hibernate.show_sql" value="true" />
```

애플리케이션을 실행하면 콘솔에 다음 DDL이 출력되면서 실제 테이블이 생성된다.

====== DDL 콘솔 출력 ======

```
Hibernate:
    drop table MEMBER if exists
Hibernate:
    create table MEMBER (
        ID varchar(255) not null,
```

```
        NAME varchar(255),
        age integer,
        roleType varchar(255),
        createdDate timestamp,
        lastModifiedDate timestamp,
        description clob,
        primary key (ID)
    )
```

실행된 결과를 보면 기존 테이블을 삭제하고 다시 생성한 것을 알 수 있다. 그리고 방금 추가한 `roleType`은 `VARCHAR` 타입으로, `createdDate`, `lastModifiedDate`는 `TIMESTAMP` 타입으로, `description`은 `CLOB` 타입으로 생성되었다.

자동 생성되는 DDL은 지정한 데이터베이스 방언에 따라 달라진다. 만약 Oracle 데이터베이스용 방언을 적용했다면 `varchar` 대신에 `varchar2` 타입이, `integer` 대신에 `number` 타입이 생성된다.

====== Oracle 데이터베이스 방언 결과 ======

```
create table MEMBER (
    ID varchar2(255 char) not null,
    NAME varchar2(10 char) not null,
    age number(10,0),
    roleType varchar2(255 char),
    createdDate timestamp,
    lastModifiedDate timestamp,
    description clob,
    primary key (ID)
)
```

스키마 자동 생성 기능을 사용하면 **애플리케이션 실행 시점에 데이터베이스 테이블이 자동으로 생성**되므로 개발자가 테이블을 직접 생성하는 수고를 덜 수 있다. 하지만 스키마 자동 생성 기능이 만든 DDL은 운영 환경에서 사용할 만큼 완벽하지는 않으므로 개발 환경에서 사용하거나 매핑을 어떻게 해야 하는지 참고하는 정도로만 사용하는 것이 좋다.

객체와 테이블을 매핑하는데 아직 익숙하지 않다면 데이터베이스 스키마 자동 생성하기를 적극적으로 활용하자. 이 기능을 사용해서 **생성된 DDL을 보면 엔티티와 테이블이 어떻게 매핑되는지 쉽게 이해할 수 있다.** 따라서 스키마 자동 생성하기는 엔티티와 테이블을 어떻게 매핑해야 하는지 알려주는 가장 훌륭한 학습 도구이다.

`hibernate.hbm2ddl.auto` 속성의 옵션은 다음과 같다.

표 - `hibernate.hbm2ddl.auto` 속성

옵션	설명
`create`	기존 테이블을 삭제하고 새로 생성한다. DROP + CREATE
`create-drop`	`create` 속성에 추가로 애플리케이션을 종료할 때 생성한 DDL을 제거한다. DROP + CREATE + DROP
`update`	데이터베이스 테이블과 엔티티 매핑정보를 비교해서 변경 사항만 수정한다.
`validate`	데이터베이스 테이블과 엔티티 매핑정보를 비교해서 차이가 있으면 경고를 남기고 애플리케이션을 실행하지 않는다. 이 설정은 DDL을 수정하지 않는다.
`none`	자동 생성 기능을 사용하지 않으려면 `hibernate.hbm2ddl.auto` 속성 자체를 삭제하거나 유효하지 않은 옵션 값을 주면 된다. (참고로 `none` 은 유효하지 않은 옵션 값이다.)

> **HBM2DDL 주의사항:** 운영 서버에서 `create`, `create-drop`, `update` 처럼 DLL을 수정하는 옵션은 절대 사용하면 안 된다. 오직 개발 서버나 개발 단계에서만 사용해야 한다. 이 옵션들은 운영 중인 데이터베이스의 테이블이나 컬럼을 삭제할 수 있다.
>
> **개발 환경에 따른 추천 전략은 다음과 같다.**
>
> - 개발 초기 단계는 `create` 또는 `update`
> - 초기화 상태로 자동화된 테스트를 진행하는 개발자 환경과 CI 서버는 `create` 또는 `create-drop`
> - 테스트 서버는 `update` 또는 `validate`
> - 스테이징과 운영 서버는 `validate` 또는 `none`

> **JPA 2.1 추가사항:** JPA는 2.1부터 스키마 자동 생성 기능을 표준으로 지원한다. 하지만 하이버네이트의 `hibernate.hbm2ddl.auto` 속성이 지원하는 `update`, `validate` 옵션을 지원하지 않는다.
>
> ```
> <property name="javax.persistence.schema-generation.database.action"
>           value="drop-and-create"/>
> ```

> 지원 옵션: `none` , `create` , `drop-and-create` , `drop`

DDL 생성 기능

회원 이름은 필수로 입력되어야 하고, 10자를 초과하면 안된다는 제약조건이 추가되었다. 스키마 자동 생성하기를 통해 만들어지는 DDL에 이 제약조건을 추가해보자.

```
@Entity
@Table(name="MEMBER")
public class Member {

    @Id
    @Column(name = "ID")
    private String id;

    @Column(name = "NAME", nullable = false, length = 10) //추가 //**
    private String username;
    ...
}
```

`@Column` 매핑정보의 `nullable` 속성 값을 `false` 로 지정하면 자동 생성되는 DDL에 `not null` 제약조건을 추가할 수 있다. 그리고 `length` 속성 값을 사용하면 자동 생성되는 DDL에 문자의 크기를 지정할 수 있다. `nullable = false` , `length = 10` 으로 지정해보자.

====== 생성된 DDL ======

```
create table MEMBER (
    ID varchar(255) not null,
    NAME varchar(10) not null,
    ...
    primary key (ID)
)
```

생성된 DDL의 `NAME` 컬럼을 보면 `not null` 제약조건이 추가되었고, `varchar(10)` 으로 문자의 크기가 10자리로 제한된 것을 확인할 수 있다.

유니크 제약조건

이번에는 유니크 제약조건을 만들어 주는 `@Table` 의 `uniqueConstraints` 속성을 알아보자.

====== 유니크 제약조건 ======

```
@Entity(name="Member")
@Table(name="MEMBER", uniqueConstraints = {@UniqueConstraint( //추가 //**
        name = "NAME_AGE_UNIQUE",
        columnNames = {"NAME", "AGE"} )})
public class Member {

    @Id
    @Column(name = "id")
    private String id;

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

    private Integer age;

    ...
```

====== 자동 생성된 DDL ======

```
ALTER TABLE MEMBER
        ADD CONSTRAINT NAME_AGE_UNIQUE  UNIQUE (NAME, AGE)
```

자동 생성된 DDL을 보면 유니크 제약조건이 추가되었다. 앞서본 `@Column`의 `length`와 `nullable` 속성을 포함해서 **이런 기능들은 단지 DDL을 자동 생성할 때만 사용되고 JPA의 실행 로직에는 영향을 주지 않는다.** 따라서 스키마 자동 생성 기능을 사용하지 않고 직접 DDL을 만든다면 사용할 이유가 없다. 그래도 이 기능을 사용하면 애플리케이션 개발자가 엔티티만 보고도 손쉽게 다양한 제약조건을 파악할 수 있는 장점이 있다.

JPA에는 이처럼 애플리케이션의 실행 동작에는 영향을 주지 않지만, 자동 생성되는 DDL을 위한 기능들도 있다. 뒤에 나오는 레퍼런스에는 이런 DDL을 위한 기능들을 별도로 구분해 두었다.

다음은 데이터베이스의 기본 키를 어떻게 매핑해야 하는지 알아보자.

기본 키(Primary Key) 매핑

```
@Entity
public class Member {
```

```
    @Id
    @Column(name = "ID")
    private String id;
    ...
}
```

지금까지 `@Id` 어노테이션만 사용해서 회원의 기본 키를 애플리케이션에서 직접 할당했다. 기본 키를 애플리케이션에서 직접 할당하는 대신에 데이터베이스가 생성해주는 값을 사용하려면 어떻게 매핑해야 할까? 예를 들어 Oracle의 시퀀스 오브젝트라던가 아니면 MySQL의 `AUTO_INCREMENT` 같은 기능을 사용해서 생성된 값을 기본 키로 사용하려면 어떻게 해야할까?

데이터베이스마다 기본 키를 생성하는 방식이 서로 다르므로 이 문제를 해결하기는 쉽지 않다. JPA는 이런 문제들을 어떻게 해결하는지 알아보자.

JPA가 제공하는 데이터베이스 기본 키 생성 전략은 다음과 같다.

- 직접 할당 : 기본 키를 애플리케이션에서 직접 할당한다.
- 자동 생성 : 대리 키 사용 방식
  - IDENTITY : 기본 키 생성을 데이터베이스에 위임한다.
  - SEQUENCE : 데이터베이스 시퀀스를 사용해서 기본 키를 할당한다.
  - TABLE : 키생성 테이블을 사용한다.

자동 생성 전략이 이렇게 다양한 이유는 데이터베이스 벤더마다 지원하는 방식이 다르기 때문이다. 예를 들어 Oracle 데이터베이스는 시퀀스를 제공하지만 MySQL은 시퀀스를 제공하지 않는다. 대신에 MySQL은 기본 키 값을 자동으로 채워주는 `AUTO_INCREMENT` 기능을 제공한다. 따라서 SEQUENCE나 IDENTITY 전략은 사용하는 데이터베이스에 의존한다. TABLE 전략은 키 생성용 테이블을 하나 만들어두고 마치 시퀀스처럼 사용하는 방법이다. 이 전략은 테이블을 활용하므로 모든 데이터베이스에서 사용할 수 있다.

기본 키를 직접 할당하려면 `@Id`만 사용하면 되고, 자동 생성 전략을 사용하려면 `@Id`에 `@GeneratedValue`를 추가하고 원하는 키 생성 전략을 선택하면 된다.

먼저 각각의 전략을 어떻게 사용하는지 알아보고 `@GeneratedValue` 어노테이션도 살펴보자.

> **주의:** 키 생성 전략을 사용하려면 `persistence.xml`에 `hibernate.id.new_generator_mappings=true` 속성을 반드시 추가해야 한다. 하이버네이트는 더 효과적이고 JPA 규격에 맞는 새로운 키 생성 전략을 개발했는데 과거 버전과의

> 호환성을 유지하려고 기본값을 `false`로 두었다. 기존 하이버네이트 시스템을 유지보수 하는 것이 아니라면 반드시 `true`로 설정하자. 지금부터 설명하는 내용도 이 옵션을 `true`로 설정했다고 가정한다.

```
<property name="hibernate.id.new_generator_mappings" value="true" />
```

- 기본 키 직접 할당 전략

```
@Id
@Column(name = "id")
private String id;
```

`@Id` 적용 가능 자바 타입: 자바 기본형, 자바 래퍼(Wrapper)형, `String`, `java.util.Date`, `java.sql.Date`, `java.math.BigDecimal`, `java.math.BigInteger`

기본 키 직접 할당 전략은 `em.persist()`로 엔티티를 저장하기 전에 애플리케이션에서 기본 키를 직접 할당하는 방법이다.

```
Board board = new Board();
board.setId("id1") //기본 키 직접 할당
em.persist(board);
```

> **참고:** 기본 키 직접 할당 전략에서 식별자 값 없이 저장하면 예외가 발생하는데, 어떤 예외가 발생하는지 JPA 표준에는 정의되어 있지 않다. 하이버네이트를 구현체로 사용하면 JPA 최상위 예외인 `javax.persistence.PersistenceException` 예외가 발생하는데, 내부에 하이버네이트 예외인 `org.hibernate.id.IdentifierGenerationException` 예외를 포함하고 있다.

- IDENTITY 전략

IDENTITY는 기본 키 생성을 데이터베이스에 위임하는 전략이다. 주로 MySQL, PostgreSQL, SQL Server, DB2에서 사용한다.

예를 들어 MySQL의 `AUTO_INCREMENT` 기능은 데이터베이스가 기본 키를 자동으로 생성해준다. 다음 예제를 보자.

====== MySQL의 AUTO_INCREMENT 기능 ======

```
CREATE TABLE BOARD (
  ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  DATA VARCHAR(255)
);

INSERT INTO BOARD(DATA) VALUES('A');
INSERT INTO BOARD(DATA) VALUES('B');
```

테이블을 생성할 때 기본 키 컬럼인 `ID`에 `AUTO_INCREMENT`를 추가했다. 이제 데이터베이스에 값을 저장할 때 `ID` 컬럼을 비워두면 데이터베이스가 순서대로 값을 채워준다.

BOARD 테이블 결과

ID	DATA
1	A
2	B

BOARD 테이블 결과를 보면 `ID` 컬럼에 자동으로 값이 입력된 것을 확인할 수 있다.

IDENTITY 전략은 지금 설명한 `AUTO_INCREMENT`를 사용한 예제처럼 데이터베이스에 값을 저장하고 나서야 기본 키 값을 구할 수 있을 때 사용한다.

개발자가 엔티티에 직접 식별자를 할당하면 `@Id` 어노테이션만 있으면 되지만 지금처럼 식별자가 생성되는 경우에는 `@GeneratedValue` 어노테이션을 사용하고 식별자 생성 전략을 선택해야 한다. IDENTITY 전략을 사용하려면 `@GeneratedValue`의 `strategy` 속성 값을 `GenerationType.IDENTITY`로 지정하면 된다. 이 전략을 사용하면 JPA는 기본 키 값을 얻어오기 위해 데이터베이스를 추가로 조회한다.

====== 매핑 코드 =====

```
@Entity
public class Board {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) //**
    private Long id;
    ...
}
```

====== 사용 코드 ======

```
private static void logic(EntityManager em) {
    Board board = new Board();
    em.persist(board);
    System.out.println("board.id = " + board.getId());
}
//출력 : board.id = 1
```

사용 코드를 보면 `em.persist()`를 호출해서 엔티티를 저장한 직후에 할당된 식별자 값을 출력했다. 출력된 값 1은 저장 시점에 데이터베이스가 생성한 값을 JPA가 조회한 것이다.

> **참고: IDENTITY 전략과 최적화**
>
> IDENTITY 전략은 데이터를 데이터베이스에 INSERT 한 후에 기본 키 값을 조회할 수 있다. 따라서 엔티티에 식별자 값을 할당하려면 JPA는 추가로 데이터베이스를 조회해야 한다. JDBC3에 추가된 `Statement.getGeneratedKeys()`를 사용하면 데이터를 저장하면서 동시에 생성된 기본 키 값도 얻어 올 수 있다. 하이버네이트는 이 메서드를 사용해서 데이터베이스와 한 번만 통신한다.

> **주의:** 엔티티가 영속 상태가 되려면 식별자가 반드시 필요하다. 그런데 IDENTITY 식별자 생성 전략은 엔티티를 데이터베이스에 저장해야 식별자를 구할 수 있으므로 `em.persist()`를 호출하는 즉시 INSERT SQL이 데이터베이스에 전달된다. 따라서 이 전략은 트랜잭션을 지원하는 쓰기 지연이 동작하지 않는다.

- SEQUENCE 전략

데이터베이스 시퀀스는 유일한 값을 순서대로 생성하는 특별한 데이터베이스 오브젝트다.
SEQUENCE 전략은 이 시퀀스를 사용해서 기본 키를 생성한다. 이 전략은 시퀀스를 지원하는 Oracle, PostgreSQL, DB2, H2 데이터베이스에서 사용할 수 있다.

시퀀스를 사용하는 예제를 보자.

====== 시퀀스 DDL ======

```
CREATE TABLE BOARD (
    ID BIGINT NOT NULL PRIMARY KEY,
    DATA VARCHAR(255)
)
CREATE SEQUENCE BOARD_SEQ START WITH 1 INCREMENT BY 1; //시퀀스 생성
```

====== 매핑 코드 ======

```
@Entity
@SequenceGenerator(
    name = "BOARD_SEQ_GENERATOR",
    sequenceName = "BOARD_SEQ", //매핑할 데이터베이스 시퀀스 이름
    initialValue = 1, allocationSize = 1)
public class Board {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "BOARD_SEQ_GENERATOR") //**
    private Long id;
    ...
}
```

우선 사용할 데이터베이스 시퀀스를 매핑해야 한다. 예제에서는 `@SequenceGenerator`를 사용해서 `BOARD_SEQ_GENERATOR`라는 시퀀스 생성기를 등록했다. 그리고 `sequenceName` 속성의 이름으로 `BOARD_SEQ`를 지정했는데 JPA는 이 시퀀스 생성기를 실제 데이터베이스의 `BOARD_SEQ` 시퀀스와 매핑한다.

다음으로 키 생성 전략을 `GenerationType.SEQUENCE`로 설정하고 `generator = "BOARD_SEQ_GENERATOR"`로 방금 등록한 시퀀스 생성기를 선택했다. 이제부터 `id` 식별자 값은 `BOARD_SEQ_GENERATOR` 시퀀스 생성기가 할당한다.

====== 사용 코드 ======

```
private static void logic(EntityManager em) {
    Board board = new Board();
    em.persist(board);
    System.out.println("board.id = " + board.getId());
```

```
}
//출력 : board.id = 1
```

사용 코드는 IDENTITY 전략과 같지만 내부 동작 방식은 다르다. SEQUENCE 전략은 `em.persist()` 를 호출할 때 먼저 데이터베이스 시퀀스를 사용해서 식별자를 조회한다. 그리고 조회한 식별자를 엔티티에 할당한 후에 엔티티를 데이터베이스에 저장한다. 반대로 이전에 설명했던 IDENTITY 전략은 먼저 엔티티를 데이터베이스에 저장한 후에 식별자를 조회해서 엔티티의 식별자에 할당한다.

– @SequenceGenerator

`@SequenceGenerator` 를 분석해보자.

속성 정리

속성	기능	기본값
name	식별자 생성기 이름	필수
sequenceName	데이터베이스에 등록되어 있는 시퀀스 이름	hibernate_sequence
initialValue	DDL 생성시에만 사용됨, 시퀀스 DDL을 생성할 때 처음 시작하는 수를 지정한다.	1
allocationSize	시퀀스 한 번 호출에 증가하는 수 (성능 최적화에 사용됨)	**50**
catalog, schema	데이터베이스 catalog,schema 이름	

매핑할 DDL

```
create sequence [sequenceName]
start with [initialValue] increment by [allocationSize]
```

JPA 표준 명세에서는 `sequenceName` 의 기본값을 JPA 구현체가 정의하도록 했다. 위에서 설명한 기본값은 하이버네이트 기준이다.

> **주의** : `SequenceGenerator.allocationSize`의 기본값이 50인 것에 주의해야 한다. JPA가 기본으로 생성하는 데이터베이스 시퀀스는 `create sequence [sequenceName] start with 1 increment by 50` 이므로 시퀀스를 호출할 때 마다 값이 50씩 증가한다. 기본값이 50인 이유는 최적화 때문인데 다음 참고에서 설명하겠다. 데이터베이스 시퀀스 값이 하나씩 증가하도록 설정되어 있으면 이 값을 반드시 `1`로 설정해야 한다.

> **참고: SEQUENCE 전략과 최적화**
>
> SEQUENCE 전략은 데이터베이스 시퀀스를 통해 식별자를 조회하는 추가 작업이 필요하다. 따라서 데이터베이스와 2번 통신한다.
>
> 1. 식별자를 구하려고 데이터베이스 시퀀스를 조회한다. 예) `SELECT BOARD_SEQ.NEXTVAL FROM DUAL`
> 2. 조회한 시퀀스를 기본 키 값으로 사용해 데이터베이스에 저장한다. 예) `INSERT INTO BOARD...`
>
> JPA는 시퀀스에 접근하는 횟수를 줄이기 위해 `@SequenceGenerator.allocationSize`를 사용한다. 간단히 설명하자면 여기에 설정한 값만큼 한 번에 시퀀스 값을 증가시키고 나서 그만큼 메모리에 시퀀스 값을 할당한다. 예를 들어 이 값이 50이면 시퀀스를 한 번에 50 증가시킨 다음에 1 ~ 50까지는 메모리에서 식별자를 할당한다. 그리고 51이 되면 시퀀스 값을 100으로 증가시킨 다음 51 ~ 100까지 메모리에서 식별자를 할당한다.
>
> 이 최적화 방법은 시퀀스 값을 선점하기 때문에 여러 JVM이 동시에 동작해도 기본 키 값이 충돌하지 않는다.

> **참고:** `@SequenceGenerator`는 다음과 같이 `@GeneratedValue` 옆에 사용해도 된다.
>
> ```
> @Entity
> public class Board {
>     @Id
>     @GeneratedValue(...)
>     @SequenceGenerator(...)
>     private Long id;
> ```

- TABLE 전략

TABLE 전략은 키 생성 전용 테이블을 하나 만들고 여기에 이름과 값으로 사용할 컬럼을 만들어 데이터베이스 시퀀스를 흉내 내는 전략이다. 이 전략은 테이블을 사용하므로 모든 데이터베이스에 적용할 수 있다.

TABLE 전략을 사용하려면 먼저 키 생성 용도로 사용할 테이블을 만들어야 한다.

====== 키 생성 DDL ======

```
create table MY_SEQUENCES (
    sequence_name varchar(255) not null ,
    next_val bigint,
    primary key ( sequence_name )
)
```

`sequence_name` 컬럼을 시퀀스 이름으로 사용하고 `next_val` 컬럼을 시퀀스 값으로 사용한다. 참고로 컬럼의 이름은 변경할 수 있는데 여기서 사용한 것이 기본값이다.

====== 매핑 코드 =====

```
@Entity
@TableGenerator(
    name = "BOARD_SEQ_GENERATOR",
    table = "MY_SEQUENCES",
    pkColumnValue = "BOARD_SEQ", allocationSize = 1)
public class Board {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE,
                    generator = "BOARD_SEQ_GENERATOR") //**
    private Long id;
```

먼저 `@TableGenerator`를 사용해서 테이블 키 생성기를 등록한다. 여기서는 `BOARD_SEQ_GENERATOR`라는 이름의 테이블 키 생성기를 등록하고 방금 생성한 `MY_SEQUENCES` 테이블을 키 생성용 테이블로 매핑했다.

다음으로 TABLE 전략을 사용하기 위해 `GenerationType.TABLE`을 선택했다. 그리고 `@GeneratedValue.generator`에 방금 만든 테이블 키 생성기를 지정했다. 이제부터 `id` 식별자 값은 `BOARD_SEQ_GENERATOR` 테이블 키 생성기가 할당한다.

====== 사용 코드 ======

```
private static void logic(EntityManager em) {
    Board board = new Board();
    em.persist(board);
    System.out.println("board.id = " + board.getId());
}
//출력 : board.id = 1
```

TABLE 전략은 시퀀스 대신에 테이블을 사용한다는 것만 제외하면 SEQUENCE 전략과 내부 동작방식이 같다.

다음 `MY_SEQUENCES` 테이블을 보면 `@TableGenerator.pkColumnValue`에서 지정한 “BOARD_SEQ”가 컬럼명으로 추가된 것을 확인할 수 있다. 이제 키 생성기를 사용할 때 마다 `next_val` 컬럼 값이 증가한다. 참고로 `MY_SEQUENCES` 테이블에 값이 없으면 JPA가 값을 INSERT 하면서 초기화 하므로 값을 미리 넣어둘 필요는 없다.

MY_SEQUENCES 테이블

sequence_name	next_val
BOARD_SEQ	2
MEMBER_SEQ	10
PRODUCT_SEQ	50
…	…

– @TableGenerator

`@TableGenerator`를 분석해보자.

속성 정리

속성	기능	기본값
name	식별자 생성기 이름	필수
table	키생성 테이블명	hibernate_sequences
pkColumnName	시퀀스 컬럼명	sequence_name
valueColumnName	시퀀스 값 컬럼명	next_val
pkColumnValue	키로 사용할 값 이름	엔티티 이름
initialValue	초기 값, 마지막으로 생성된 값이 기준이다.	0
allocationSize	시퀀스 한 번 호출에 증가하는 수 (성능 최적화에 사용됨)	50
catalog, schema	데이터베이스 catalog,schema 이름	
uniqueConstraints(DDL)	유니크 제약 조건을 지정할 수 있다.	

JPA 표준 명세에서는 `table`, `pkColumnName`, `valueColumnName`의 기본값을 JPA 구현체가 정의하도록 했다. 위에서 설명한 기본값은 하이버네이트 기준이다.

====== 매핑할 DDL ======

{table}

{pkColumnName}	{valueColumnName}
{pkColumnValue}	{initialValue}

> **참고: TABLE 전략과 최적화**
>
> TABLE 전략은 값을 조회하면서 SELECT 쿼리를 사용하고 다음 값으로 증가시키기 위해 UPDATE 쿼리를 사용한다. 이 전략은 SEQUENCE 전략과 비교해서 데이터베이스와 한 번 더 통신하는 단점이 있다. TABLE 전략을 최적화 하려면 `@TableGenerator.allocationSize`를 사용하면 된다. 이 값을 사용해서 최적화하는 방법은 SEQUENCE 전략과 같다.

- AUTO 전략

데이터베이스의 종류도 많고 기본 키를 만드는 방법도 다양하다. `GenerationType.AUTO`는 선택한 데이터베이스 방언에 따라 `IDENTITY`, `SEQUENCE`, `TABLE` 전략 중 하나를 자동으로 선택한다. 예를 들어 Oracle을 선택하면 `SEQUENCE`를, MySQL을 선택하면 `IDENTITY`를 사용한다.

====== 매핑 코드 =====

```
@Entity
public class Board {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    ...
}
```

`@GeneratedValue.strategy`의 기본값은 `AUTO`다. 따라서 다음과 같이 사용해도 결과는 같다.

```
@Id @GeneratedValue
private Long id;
```

AUTO 전략의 장점은 데이터베이스를 변경해도 코드를 수정할 필요가 없다는 것이다. 특히 키 생성 전략이 아직 확정되지 않은 개발 초기 단계나 프로토타입 개발시 편리하게 사용할 수 있다.

AUTO를 사용할 때 SEQUENCE나 TABLE 전략이 선택되면 시퀀스나 키 생성용 테이블을 미리 만들어 두어야 한다. 만약 스키마 자동 생성 기능을 사용한다면 하이버네이트가 기본값을 사용해서 적절한 시퀀스나 키 생성용 테이블을 만들어 줄 것이다.

기본 키 매핑 정리

영속성 컨텍스트는 엔티티를 식별자 값으로 구분하므로 엔티티를 영속 상태로 만들려면 식별자 값이 반드시 있어야 한다. `em.persist()`를 호출한 직후에 발생하는 일을 식별자 할당 전략별로 정리하면 다음과 같다

- 직접 할당 : `em.persist()`를 호출하기 전에 애플리케이션에서 직접 식별자 값을 할당해야 한다. 만약 식별자 값이 없으면 예외가 발생한다.
- `SEQUENCE` : 데이터베이스 시퀀스에서 식별자 값을 획득한 후 영속성 컨텍스트에 저장한다.
- `TABLE` : 데이터베이스 시퀀스 생성용 테이블에서 식별자 값을 획득한 후 영속성 컨텍스트에 저장

한다.

- `IDENTITY` : 데이터베이스에 엔티티를 저장해서 식별자 값을 획득한 후 영속성 컨텍스트에 저장한다. (`IDENTITY` 전략은 테이블에 데이터를 저장해야 식별자 값을 획득할 수 있다.)

- [참고] 권장하는 식별자 선택 전략

> **참고: 권장하는 식별자 선택 전략**
>
> 데이터베이스 기본 키는 다음 3가지 조건을 모두 만족해야 한다.
>
> 1. `null` 값은 허용하지 않는다.
> 2. 유일해야 한다.
> 3. 변해선 안 된다.
>
> 테이블의 기본 키를 선택하는 전략은 크게 2가지가 있다.
>
> - **자연 키**(natural key)
>   - 비즈니스에 의미가 있는 키
>   - 예 : 주민등록번호, E-MAIL, 전화번호
>
> - **대리 키**(surrogate key)
>   - 비즈니스와 관련 없는 임의로 만들어진 키, 대체 키로도 불린다.
>   - 예 : Oracle 시퀀스, `auto_increment` , 키생성 테이블 사용
>
> **자연 키보다는 대리 키를 권장한다.**
>
> 자연 키와 대리 키는 일장 일단이 있지만 될 수 있으면 대리 키의 사용을 권장한다. 예를 들어 자연 키인 전화번호를 기본 키로 선택한다면 그 번호가 유일할 수는 있지만, 전화번호가 없을 수도 있고 전화번호가 변경될 수도 있다. 따라서 기본 키로 적당하지 않다. 문제는 주민등록번호처럼 그럴듯하게 보이는 값이다. 이 값은 `null` 이 아니고 유일하며 변하지 않는다는 3가지 조건을 모두 만족하는 것 같다. 하지만 현실과 비즈니스 규칙은 생각보다 쉽게 변한다. 주민등록번호 조차도 여러 가지 이유로 변경될 수 있다.
>
> **비즈니스 환경은 언젠가 변한다.**
>
> 나의 경험을 하나 이야기하겠다. 레거시 시스템을 유지보수 할 일이 있었는데, 분석해보니 회원 테이블에 주민등록번호가 기본 키로 잡혀 있었다. 회원과 관련된 수많은 테이블에서 조인을 위해 주민등록번호를 외래 키로 가지고 있었고 심지어 자식 테이블에 자식 테이블까지

> 주민등록번호가 내려가 있었다. 문제는 정부 정책이 변경되면서 법적으로 주민등록번호를 저장할 수 없게 되면서 발생했다. 결국 데이터베이스 테이블은 물론이고 수많은 애플리케이션 로직을 수정했다. 만약 데이터베이스를 처음 설계할 때부터 자연 키인 주민등록번호 대신에 비즈니스와 관련 없는 대리 키를 사용했다면 수정할 부분이 많지는 않았을 것이다.
>
> 기본 키의 조건을 현재는 물론이고 미래까지 충족하는 자연 키를 찾기는 쉽지 않다. 대리 키는 비즈니스와 무관한 임의의 값이므로 요구사항이 변경되어도 기본 키가 변경되는 일은 드물다. 대리 키를 기본 키로 사용하되 주민등록번호나 E-MAIL처럼 자연 키의 후보가 되는 컬럼들은 필요에 따라 유니크 인덱스를 설정해서 사용하는 것을 권장한다.
>
> **JPA는 모든 엔티티에 일관된 방식으로 대리 키 사용을 권장한다.**
>
> 비즈니스 요구사항은 계속해서 변하는데 테이블은 한번 정의하면 변경하기 어렵다. 그런 면에서 외부 풍파에 쉽게 흔들리지 않는 대리 키가 일반적으로 좋은 선택이라 생각한다.

> **참고:** 2개 이상의 컬럼으로 기본 키를 구성하는 복합 키는 “8. 연관관계 매핑 심화2” 장에서 다루겠다.

> **주의:** 기본 키는 변하면 안 된다는 기본 원칙으로 인해, 저장된 엔티티의 기본 키 값은 절대 변경하면 안 된다. 이 경우 JPA는 예외를 발생시키거나 정상 동작하지 않는다. 식별자 자동 생성 전략을 사용하면 `setId()` 같이 식별자를 수정하는 메서드를 외부에 공개하지 않는 것도 문제를 예방하는 하나의 방법이 될 수 있다.

필드와 컬럼 매핑 - 레퍼런스

JPA가 제공하는 필드와 컬럼 매핑용 어노테이션들을 레퍼런스 형식으로 정리해보았다. 책을 학습할 때는 간단히 훑어보고, 필요한 매핑을 사용할 일이 있을 때 찾아서 자세히 읽어보는 것을 권장한다.

분류	매핑 어노테이션	설명
필드와 컬럼 매핑	`@Column`	컬럼을 매핑한다.
	`@Enumerated`	자바의 `enum` 타입을 매핑한다.
	`@Temporal`	날짜 타입을 매핑한다.
	`@Lob`	`BLOB` , `CLOB` 타입을 매핑한다.
	`@Transient`	특정 필드를 데이터베이스에 매핑하지 않는다.
기타	`@Access`	JPA가 엔티티에 접근하는 방식을 지정한다.

- @Column

객체 필드를 테이블 컬럼에 매핑한다. 가장 많이 사용되고 기능도 많다. 속성 중에 `name` , `nullable` 이 주로 사용되고 나머지는 잘 사용되지 않는 편이다.

`insertable` , `updatable` 속성은 데이터베이스에 저장되어 있는 정보를 읽기만 하고 실수로 변경하는 것을 방지하고 싶을 때 사용한다.

속성 정리

속성	기능	기본값
name	필드와 매핑할 테이블의 컬럼 이름	객체의 필드 이름
insertable (거의 사용하지 않음)	엔티티 저장시 이 필드도 같이 저장한다. `false` 로 설정하면 이 필드는 데이터베이스에 저장하지 않는다. `false` 옵션은 읽기 전용일 때 사용한다.	`true`
updatable (거의 사용하지 않음)	엔티티 수정시 이 필드도 같이 수정한다. `false` 로 설정하면 데이터베이스에 수정하지 않는다. `false` 옵션은 읽기 전용일 때 사용한다.	`true`
table	하나의 엔티티를 두 개 이상의 테이블에 매핑할 때 사용한다. 지정한 필드를 다른 테이블에	현재 클래스가 매

(거의 사용하지 않음)	매핑할 수 있다. 자세한 사용법은 “8. 연관관계 매핑 - 심화2”의 “엔티티 하나에 여러 테이블 매핑하기”에서 다룬다.	핑된 테이블
nullable(DDL)	`null` 값의 허용 여부를 설정한다. `false`로 설정하면 DDL 생성시에 `not null` 제약조건이 붙는다.	`true`
unique(DDL)	`@Table`의 `uniqueConstraints`와 같지만 한 컬럼에 간단히 유니크 제약조건을 걸 때 사용한다. 만약 두 컬럼 이상을 사용해서 유니크 제약조건을 사용하려면 클래스 레벨에서 `@Table.uniqueConstraints`를 사용해야 한다.	
columnDefinition(DDL)	데이터베이스 컬럼 정보를 직접 줄 수 있다.	필드의 자바 타입과 방언 정보를 사용해서 적절한 컬럼 타입을 생성한다.
length(DDL)	문자 길이 제약조건, `String` 타입에만 사용한다.	255
precision, scale(DDL)	`BigDecimal` 타입에서 사용한다. (`BigInteger`도 사용할 수 있다.) `precision`은 소수점을 포함한 전체 자릿수를, `scale`은 소수의 자릿수다. 참고로 `double`, `float` 타입에는 적용되지 않는다. 아주 큰 숫자나 정밀한 소수를 다루어야 할 때만 사용한다.	`precision=19`, `scale=2`

DDL 생성 속성에 따라 어떤 DDL이 생성되는지 확인해보자.

nullable (DDL 생성 기능)

```
@Column(nullable = false)
private String data;
```

```
//생성된 DDL
data varchar(255) not null
```

unique (DDL 생성 기능)

```
@Column(unique = true)
private String username;

//생성된 DDL
alter table Tablename
    add constraint UK_Xxx  unique (username)
```

columnDefinition (DDL 생성 기능)

```
@Column(columnDefinition = "varchar(100) default 'EMPTY'")
private String data;

//생성된 DDL
data varchar(100) default 'EMPTY'
```

length (DDL 생성 기능)

```
@Column(length = 400)
private String data;

//생성된 DDL
data varchar(400)
```

precision, scale (DDL 생성 기능)

```
@Column(precision = 10, scale = 2)
private BigDecimal cal;

//생성된 DDL
cal numeric(10,2) //H2, PostgreSQL
cal number(10,2)  //Oracle
cal decimal(10,2) //MySQL
```

> **참고:** `@Column` 생략
>
> `@Column`을 생략하게 되면 어떻게 될까? 대부분 `@Column` 속성의 기본값이 적용되는데, 자바 기본 타입일 때는 `nullable` 속성에 예외가 있다. 우선 코드를 보자.
>
> ```
> int data1; // @Column 생략, 자바 기본 타입
> data1 integer not null //생성된 DDL
>
> Integer data2; // @Column 생략, 객체 타입
> data2 integer //생성된 DDL
>
> @Column
> int data3; // @Column 사용, 자바 기본 타입
> data3 integer //생성된 DDL
> ```
>
> `int data1` 같은 자바 기본 타입에는 `null` 값을 입력할 수 없다. `Integer data2`처럼 객체 타입일 때만 `null` 값이 허용된다. 따라서 자바 기본 타입인 `int data1`을 DDL로 생성할 때는 `not null` 제약조건을 추가하는 것이 안전하다.
>
> JPA는 이런 상황을 고려해서 DDL 생성 기능을 사용할 때 `int data1` 같은 기본 타입에는 `not null` 제약조건을 추가한다. 반면에 `Integer data2`처럼 객체 타입이면 `null`이 입력 될 수 있으므로 `not null` 제약조건을 설정하지 않는다.
>
> 그런데 `int data3`처럼 `@Column`을 사용하면 `@Column`은 `nullable = true`가 기본값이므로 `not null` 제약조건을 설정하지 않는다. 따라서 자바 기본 타입에 `@Column`을 사용하면 `nullable = false`로 지정하는 것이 안전하다.

- @Enumerated

자바의 `enum` 타입을 매핑할 때 사용한다.

속성 정리

속성	기능	기본값
value	- `EnumType.ORDINAL` : `enum` 순서를 데이터베이스에 저장 - `EnumType.STRING` : `enum` 이름을 데이터베이스에 저장	`EnumType.ORDINAL`

`@Enumerated` 사용 예

==enum 클래스==

```
enum RoleType {
    ADMIN, USER
}
```

==enum 이름으로 매핑==

```
@Enumerated(EnumType.STRING)
private RoleType roleType;
```

==사용 코드==

```
member.setRoleType(RoleType.ADMIN);// -> DB에 문자 ADMIN으로 저장됨
```

`@Enumerated`를 사용하면 편리하게 `enum` 타입을 데이터베이스에 저장할 수 있다.

- `EnumType.ORDINAL`은 `enum`에 정의된 순서대로 `ADMIN`은 0, `USER`는 1 값이 데이터베이스에 저장된다.
  - 장점 : 데이터베이스에 저장되는 데이터 크기가 작다.
  - 단점 : 이미 저장된 `enum`의 순서를 변경할 수 없다.

- `EnumType.STRING`은 `enum` 이름 그대로 `ADMIN`은 'ADMIN', `USER`는 'USER'로 데이터베이스에 저장된다.
  - 장점 : 저장된 `enum`의 순서가 바뀌거나 `enum`이 추가되어도 안전하다.
  - 단점 : 데이터베이스에 저장되는 데이터 크기가 `ORDINAL`에 비해서 크다.

> **주의:** 기본값인 `ORDINAL`은 주의해서 사용해야 한다.
>
> ADMIN(0번), USER(1번) 사이에 enum이 하나 추가되서 ADMIN(0번), NEW(1번), USER(2번)로 설정되면 이제부터 USER는 2로 저장 되지만 기존에 데이터베이스에 저장된 값은 여전히 1로 남아있다. 따라서 이런 문제가 발생하지 않는 **`EnumType.STRING`을 권장한다.**

- @Temporal

날짜 타입(`java.util.Date` , `java.util.Calendar`)을 매핑할 때 사용한다.

속성 정리

속성	기능	기본값
value	- `TemporalType.DATE` : 날짜, 데이터베이스 `date` 타입과 매핑 (예: 2013–10–11) - `TemporalType.TIME` : 시간, 데이터베이스 `time` 타입과 매핑 (예: 11:11:11) - `TemporalType.TIMESTAMP` : 날짜와 시간, 데이터베이스 `timestamp` 타입과 매핑 (예: 2013–10–11 11:11:11)	`TemporalType` 은 필수로 지정해야한다.

`@Temporal` 사용 예

```
@Temporal(TemporalType.DATE)
private Date date; //날짜

@Temporal(TemporalType.TIME)
private Date time; //시간

@Temporal(TemporalType.TIMESTAMP)
private Date timestamp; //날짜와 시간

//==생성된 DDL==//
date date,
time time,
timestamp timestamp,
```

자바의 `Date` 타입에는 년월일 시분초가 있지만 데이터베이스에는 `date` (날짜), `time` (시간), `timestamp` (날짜와 시간)라는 세 가지 타입이 별도로 존재한다.

`@Temporal` 을 생략하면 자바의 `Date` 와 가장 유사한 `timestamp` 로 정의된다. 하지만 `timestamp` 대신에 `datetime` 을 예약어로 사용하는 데이터베이스도 있는데 데이터베이스 방언 덕분에 애플리케이션 코드는 변경하지 않아도 된다.

데이터베이스 방언에 따라 생성되는 DDL

- `datetime` : MySQL
- `timestamp` : H2, Oracle, PostgreSQL

- @Lob

데이터베이스 `BLOB`, `CLOB` 타입과 매핑한다.

속성 정리

`@Lob`에는 지정할 수 있는 속성이 없다. 대신에 매핑하는 필드 타입이 문자면 `CLOB`으로 매핑하고 나머지는 `BLOB`으로 매핑한다.

- CLOB : `String`, `char[]`, `java.sql.CLOB`
- BLOB : `byte[]`, `java.sql.BLOB`

`@Lob` 사용 예

```
@Lob
private String lobString;

@Lob
private byte[] lobByte;
```

생성된 DDL

```
//ORACLE
lobString clob,
lobByte blob,

//MySQL
lobString longtext,
lobByte longblob,

//PostgreSQL
lobString text,
lobByte oid,
```

- @Transient

이 필드는 매핑하지 않는다. 따라서 데이터베이스에 저장하지 않고 조회하지도 않는다. 객체에 임시로 어떤 값을 보관하고 싶을 때 사용할 수 있다.

```
@Transient
private Integer temp;
```

- @Access

JPA가 엔티티 데이터에 접근하는 방식을 지정한다.

- **필드 접근**: `AccessType.FIELD`로 지정한다. 필드에 직접 접근한다. 필드 접근 권한이 `private`이어도 접근할 수 있다.
- **프로퍼티 접근**: `AccessType.PROPERTY`로 지정한다. 접근자(Getter)를 사용한다.

`@Access`를 설정하지 않으면 `@Id`의 위치를 기준으로 접근 방식이 설정된다.

==== 필드 접근 ====

```
@Entity
@Access(AccessType.FIELD) //**
public class Member {

    @Id
    private String id;

    private String data1;
    private String data2;
    ...
}
```

`@Id`가 필드에 있으므로 `@Access(AccessType.FIELD)`로 설정한 것과 같다. 따라서 `@Access`는 생략해도 된다.

==== 프로퍼티 접근 ====

```
@Entity
@Access(AccessType.PROPERTY) //**
public class Member {

    private String id;
```

```
    private String data1;
    private String data2;

    @Id
    public String getId() {
        return id;
    }

    @Column
    public String getData1() {
        return data1;
    }

    public String getData2() {
        return data2;
    }
}
```

`@Id`가 프로퍼티에 있으므로 `@Access(AccessType.PROPERTY)`로 설정한 것과 같다. 따라서 `@Access`는 생략해도 된다.

필드 접근 방식과 프로퍼티 접근 방식을 함께 사용할 수도 있다.

==== 함께 사용 ====

```
@Entity
public class Member {

    @Id
    private String id;

    @Transient
    private String firstName;

    @Transient
    private String lastName;

    @Access(AccessType.PROPERTY) //**
    public String getFullName() {
        return firstName + lastName;
    }
    ...
}
```

`@Id`가 필드에 있으므로 기본은 필드 접근 방식을 사용하고 `getFullName()`만 프로퍼티 접근 방식을 사용한다. 따라서 회원 엔티티를 저장하면 회원 테이블의 `FULLNAME` 컬럼에 `firstName + lastName`의 결과가 저장된다.

[실전 예제] - 1. 요구사항 분석과 기본 매핑

작은 쇼핑몰을 만들어가면서 JPA로 실제 도메인 모델을 어떻게 구성하고 객체와 테이블을 어떻게 매핑해야 하는지 알아보자.

실전 예제는 학습한 내용을 도메인 모델에 적용하면서 점점 설계를 완성해 나갈 것이다. 그리고 완성된 도메인 모델로 웹 애플리케이션 만들기 장에서 실제 웹 애플리케이션을 만들어보겠다. 직접 코딩하면서 예제를 순서대로 따라오는 것을 권장한다.

먼저 요구사항을 분석하고 도메인 모델과 테이블을 설계하자.

> **예제 코드 : `ch04-model1`**

요구사항 분석

핵심 요구사항은 다음과 같다.

- 회원은 상품을 주문할 수 있다.
- 주문시 여러 종류의 상품을 선택할 수 있다.

요구사항을 분석해서 만든 메인 화면과 기능은 다음과 같다.

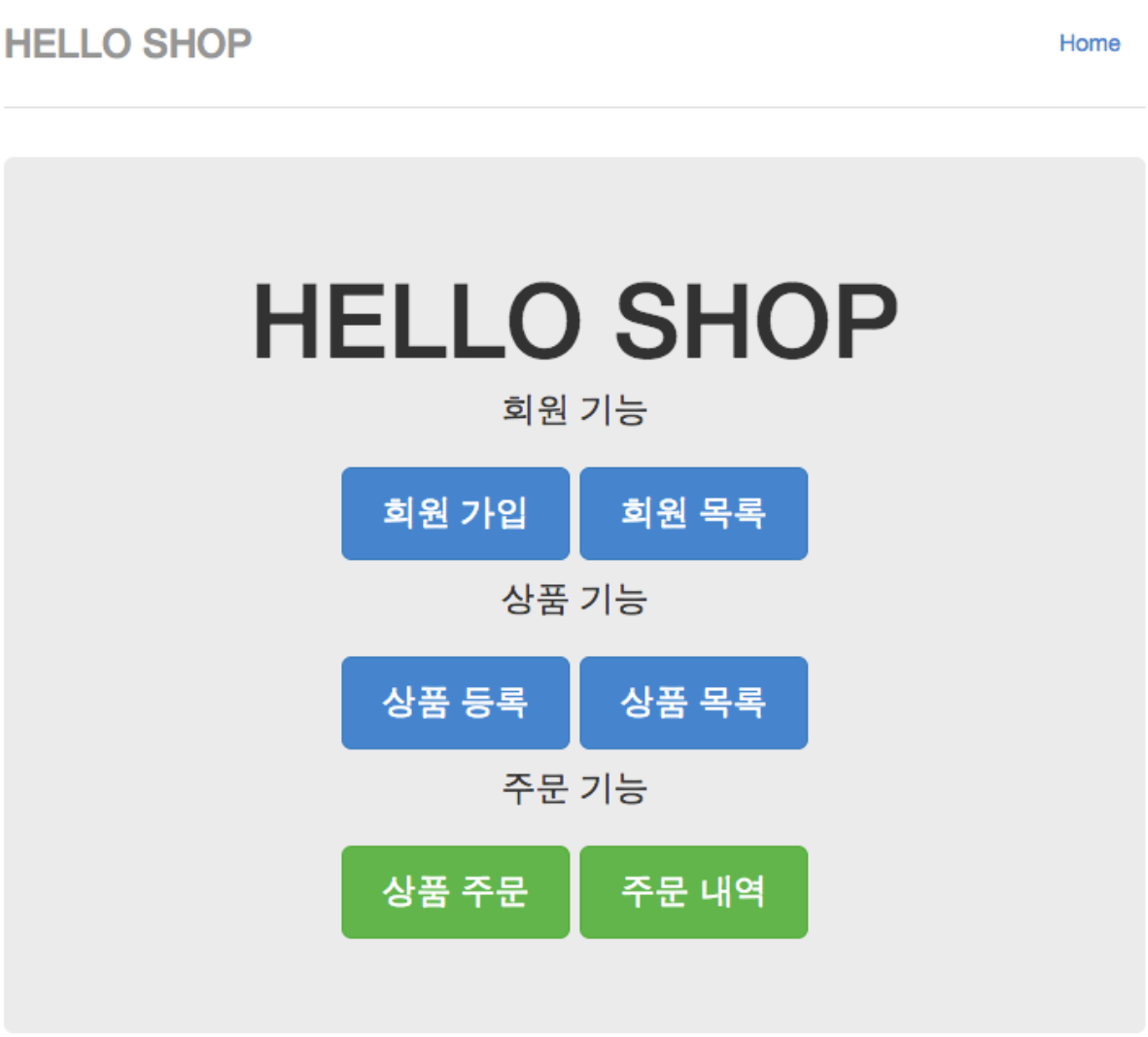

그림 - 메인 화면

- 회원 기능
  - 회원 등록
  - 회원 조회

- 상품 기능
  - 상품 등록
  - 상품 수정
  - 상품 조회

- 주문 기능
  - 상품 주문
  - 주문 내역 조회
  - 주문 취소

도메인 모델 분석

요구사항을 분석해보니 회원, 주문, 상품, 그리고 주문상품이라는 엔티티가 도출되었다.

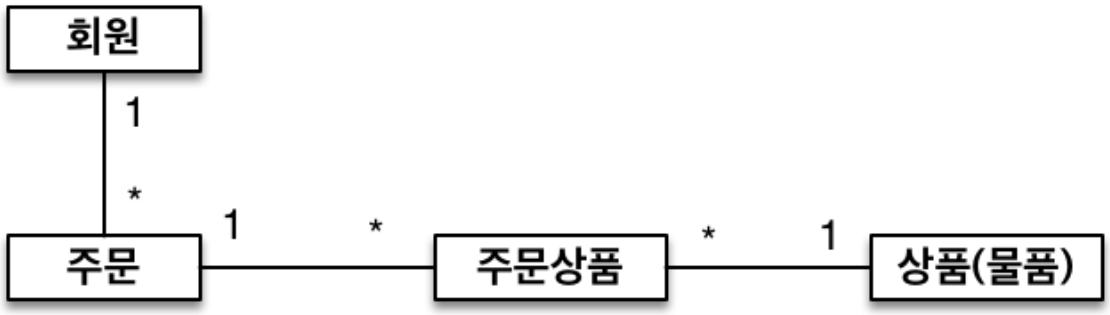

그림 - UML1

회원과 주문의 관계: 회원은 여러 번 주문할 수 있으므로 회원과 주문은 일대다 관계다.

주문과 상품의 관계: 주문할 때 여러 상품을 함께 선택할 수 있고, 같은 상품도 여러 번 주문될 수 있으므로 둘은 다대다 관계다. 하지만 이런 다대다 관계는 관계형 데이터베이스는 물론이고 엔티티에서도 거의 사용하지 않는다. 따라서 주문상품이라는 연결 엔티티를 추가해서 다대다 관계를 일대다, 다대일 관계로 풀어냈다. 그리고 주문상품에는 해당 상품을 구매한 금액과 수량 정보가 포함되어 있다.

요구사항을 분석해서 데이터베이스 테이블을 만들자.

테이블 설계

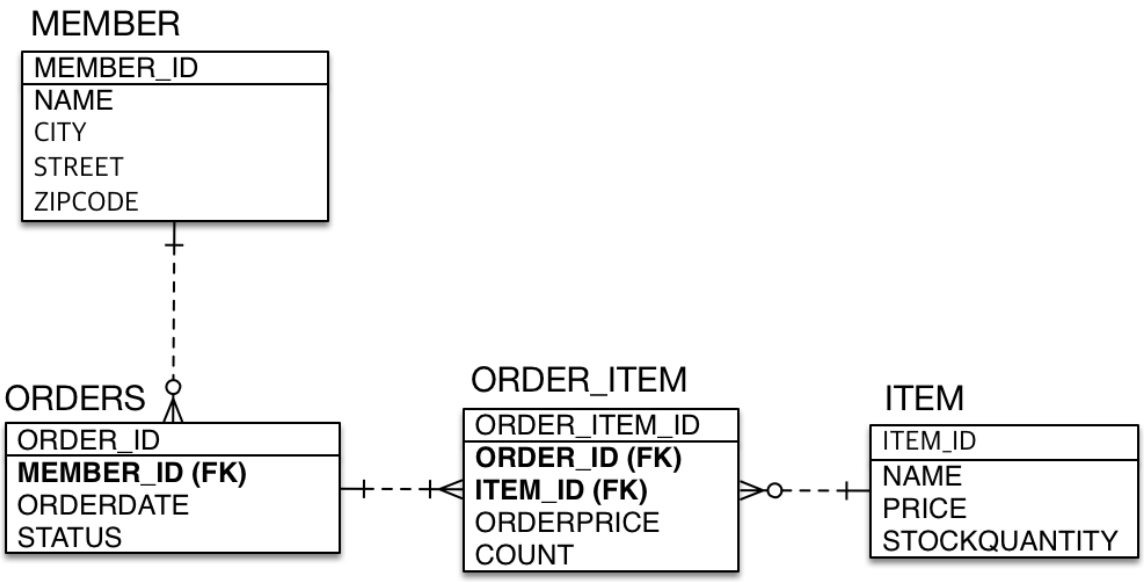

그림 - ERD1

ERD를 분석해보자.

회원(`MEMBER`): 이름(`NAME`)과 주소 정보를 가진다. 주소는 `CITY` , `STREET` , `ZIPCODE` 로 표현한다.

주문(`ORDERS`): 상품을 주문한 회원(`MEMBER_ID`)을 외래 키로 가진다. 그리고 주문 날짜(`ORDERDATE`)와 주문 상태(`STATUS`)를 가진다. 주문 상태는 주문(`ORDER`)과 취소(`CANCEL`)을 표현할 수 있다.

주문상품(`ORDER_ITEM`): 주문(`ORDER_ID`)과 주문한 상품(`ITEM_ID`)을 외래 키로 가진다. 주문 금액(`ORDERPRICE`), 주문 수량(`COUNT`) 정보를 가진다.

상품(`ITEM`): 이름(`NAME`), 가격(`PRICE`), 재고수량(`STOCKQUANTITY`)을 가진다. 상품을 주문하면 재고수량이 줄어든다.

이렇게 설계한 테이블을 기반으로 엔티티를 만들어보자.

엔티티 설계와 매핑

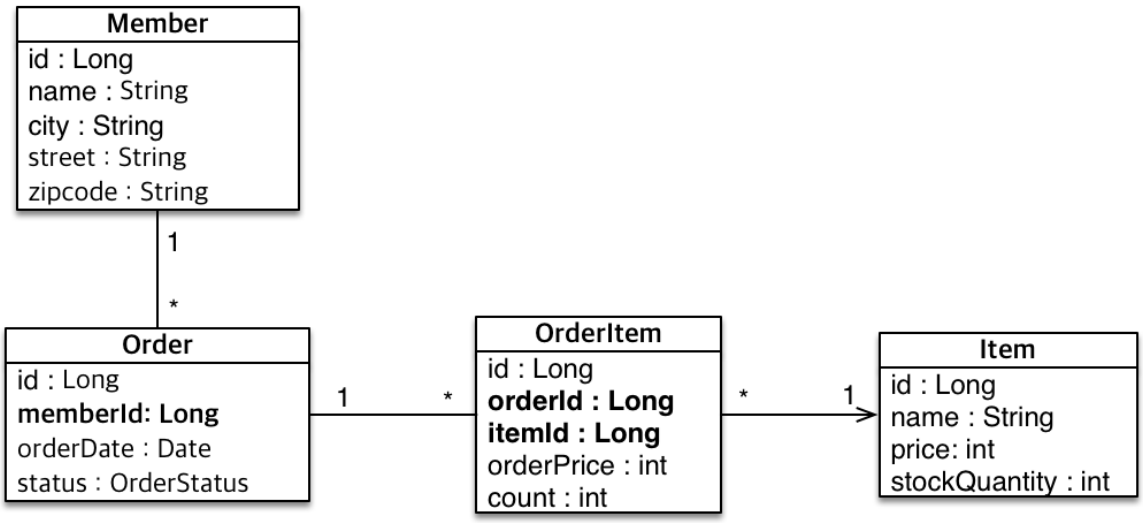

그림 - UML 상세1

=== 회원(`Member`) ===

```
package jpabook.model.entity;

import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
public class Member {

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

    private String name;

    private String city;
    private String street;
    private String zipcode;

    //Getter, Setter
    ...
}
```

회원은 이름(`name`)과 주소 정보를 가진다. 주소는 `city` , `street` , `zipcode` 로 표현한다.

식별자는 `@Id`와 `@GeneratedValue`를 사용해서 데이터베이스에서 자동 생성되도록 했다. `@GeneratedValue`의 기본 생성 전략은 `AUTO`이므로 선택한 데이터베이스 방언에 따라 `IDENTITY`, `SEQUENCE`, `TABLE`중 하나가 선택된다. 예제에서는 H2 데이터베이스를 사용하는데 이 데이터베이스는 `SEQUENCE`를 사용한다. 다른 엔티티들에 대해서도 같은 키 생성 전략을 사용하겠다.

=== 주문(`Order`) ===

```
package jpabook.model.entity;

import javax.persistence.*;
import java.util.Date;

@Entity
@Table(name = "ORDERS")
public class Order {

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

    @Column(name = "MEMBER_ID")
    private Long memberId;

    @Temporal(TemporalType.TIMESTAMP)
    private Date orderDate;     //주문 날짜 <1>

    @Enumerated(EnumType.STRING)
    private OrderStatus status;//주문 상태 <2>

    //Getter, Setter
    ...
}
```

=== OrderStatus ===

```
public enum OrderStatus {
    ORDER, CANCEL
}
```

주문은 상품을 주문한 회원(`memberId`)의 외래 키 값과 주문 날짜(`orderDate`), 주문 상태(`status`)를 가진다.

<1> - 주문 날짜는 `Date` 를 사용하고 년월일 시분초를 모두 사용하므로 `@Temporal` 에 `TemporalType.TIMESTAMP` 속성을 사용해서 매핑했다. 참고로 `@Temporal` 을 생략하면 `@Temporal(TemporalType.TIMESTAMP)` 와 같으므로 예제에서는 생략해도 된다.

<2> - 주문 상태는 열거형을 사용하므로 `@Enumerated` 로 매핑했고, `EnumType.STRING` 속성을 지정해서 열거형의 이름이 그대로 저장되도록 했다. 그리고 `OrderStatus` 열거형을 사용하므로 주문(`ORDER`)과 취소(`CANCEL`)을 표현할 수 있다.

=== 주문상품(`OrderItem`) ===

```
@Entity
@Table(name = "ORDER_ITEM")
public class OrderItem {

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

    @Column(name = "ITEM_ID")
    private Long itemId;
    @Column(name = "ORDER_ID")
    private Long orderId;

    private int orderPrice; //주문 가격
    private int count;      //주문 수량

    //Getter, Setter
    ...
}
```

주문상품은 주문(`orderId`)의 외래 키 값과 주문한 상품(`itemId`)의 외래 키 값을 가진다. 그리고 주문 금액(`orderPrice`)과 주문 수량(`count`) 정보를 가진다.

=== 상품(`Item`) ===

```
@Entity
public class Item {

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

    private String name;        //이름
    private int price;          //가격
```

```
    private int stockQuantity;   //재고수량

    //Getter, Setter
    ...
}
```

상품은 이름(`name`), 가격(`price`), 재고수량(`stockQuantity`) 정보를 가진다.

데이터 중심 설계의 문제점

이 예제의 엔티티 설계가 이상하다는 생각이 들었으면 객체 지향 설계를 의식하는 개발자고, 그렇지 않고 자연스러웠다면 데이터 중심의 개발자일 것이다.

지금 이 방식은 객체 설계를 테이블 설계에 맞춘 방법이다. 특히 테이블의 외래 키를 객체에 그대로 가져온 부분이 문제다. 왜냐하면, 관계형 데이터베이스는 연관된 객체를 찾을 때 외래 키를 사용해서 조인하면 되지만 객체에는 조인이라는 기능이 없다. 객체는 연관된 객체를 찾을 때 참조를 사용해야 한다.

설계한 엔티티로 데이터베이스 스키마 자동 생성하기를 실행해보면 ERD에 나온대로 테이블이 생성된다. 하지만 객체에서 참조 대신에 데이터베이스의 외래 키를 그대로 가지고 있으므로 `order.getMember()` 처럼 객체 그래프를 탐색할 수 없고 객체의 특성도 살릴 수 없다. 그리고 객체가 다른 객체를 참조하지도 않으므로 UML도 잘못되었다. 객체는 외래 키 대신에 참조를 사용해야 한다.

이렇게 외래 키만 가지고 있으면 연관된 엔티티를 찾을 때 외래 키로 데이터베이스를 다시 조회해야 한다. 예를 들어 주문을 조회한 다음 주문과 연관된 회원을 조회하려면 다음처럼 한번 더 조회해야 한다.

```
Order order = em.find(Order.class, orderId);
Member member = em.find(Member.class, order.getMemberId()); //외래 키로 다시 조회
```

객체는 참조를 사용해서 연관관계를 조회할 수 있다. 따라서 다음처럼 참조를 사용하는 것이 객체 지향적인 방법이다.

```
Order order = em.find(Order.class, orderId);
Member member = order.getMember(); //참조 사용
```

정리하자면 **객체는 참조**를 사용해서 연관된 객체를 찾고 **테이블은 외래 키**를 사용해서 연관된 테이블을 찾으므로 둘 사이에는 큰 차이가 있다.

JPA는 **객체의 참조와 테이블의 외래 키를 매핑**해서 객체에서는 참조를 사용하고 테이블에서는 외래 키를 사용할 수 있도록 한다. 다음 연관관계 매핑장을 통해 참조와 외래 키를 어떻게 매핑하는지 알아보자.