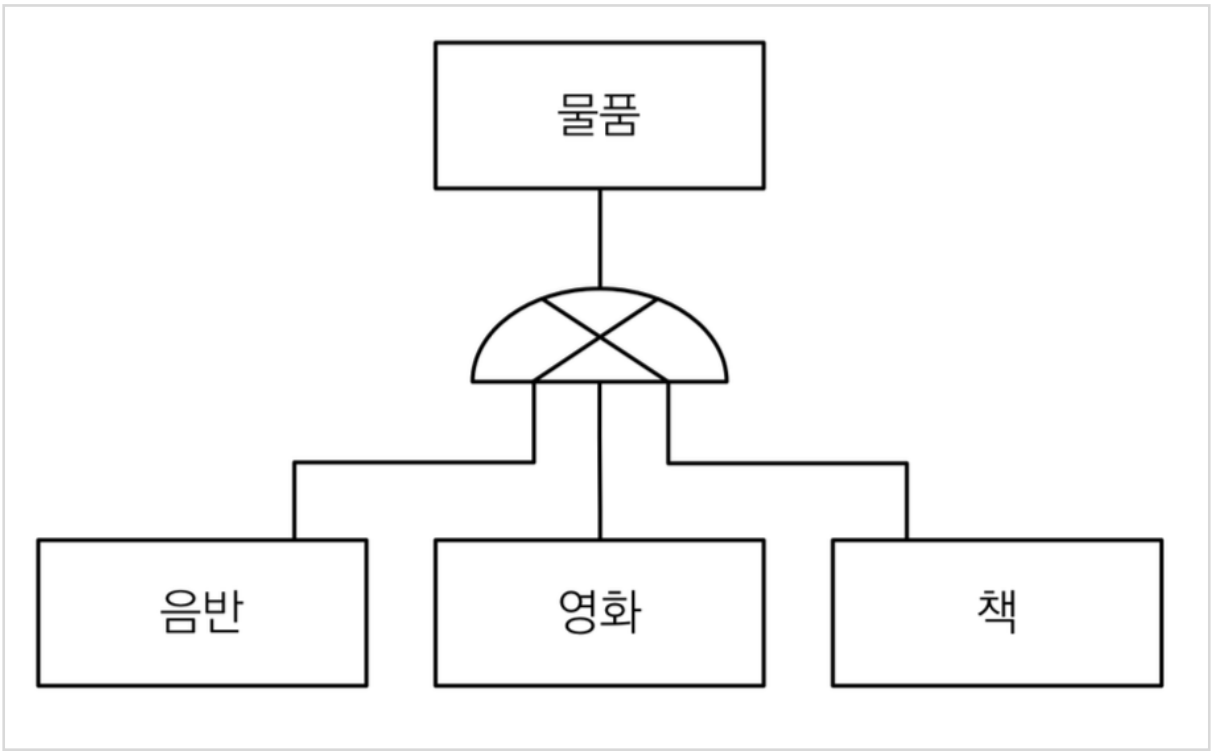
8-0. 상속관계 매핑

객체에는 상속관계가 있으나, RDB에는 상속관계가 없다.
하지만 RDB에는 슈퍼타입 / 서브타입 관계라는 모델링 기법이 있고 이것이 상속과 비슷하다.
관계형 DB 설계 - 물리 모델 / 논리 모델



위 그림이 논리 모델.
슈퍼타입 서브타입 논리 모델을 실제로 물리 모델로 구체화하여 만들려면 세가지 방법.
=> 세가지방법중 어떤 것으로 구현을 하던지 모두 지원해준다.

```
package hellojpa;

import javax.persistence.Entity;

@Entity
public class Album extends Item {

  private String artist;

}
```

```
package hellojpa;

import javax.persistence.Entity;

@Entity
public class Movie extends Item{

  private String director;
  private String actor;


}
```

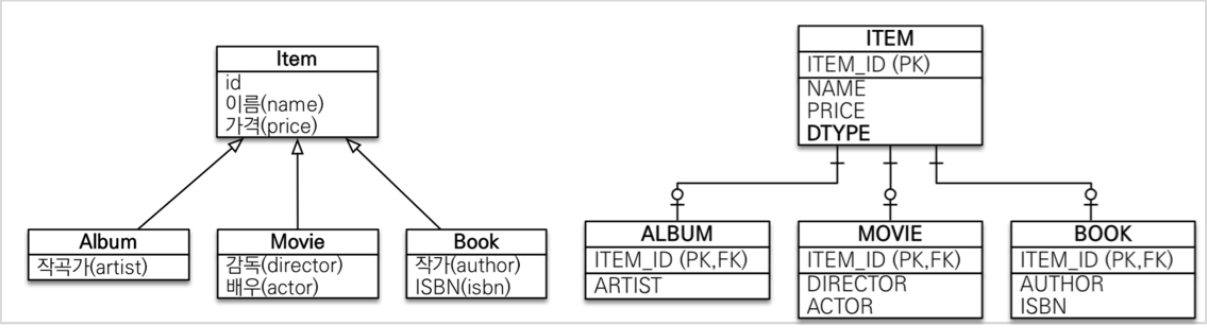
```
package hellojpa;

import javax.persistence.Entity;

@Entity
public class Book extends Item {

  private String author;
  private String ISBN;

}
```

```
package hellojpa;

import javax.persistence.Entity;

@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Item {

  @Id @GeneratedValue
```

```
    private Long id;
    private String name;
    private int price;


}
```

=> 이렇게 세팅한 다음 그냥 실행하게 되면 단일 테이블 전략으로 매핑된다.
즉, jpa의 기본 전략 자체가 단일 테이블 전략..으로 매핑이 된다.

조인 전략



아이템이라는 테이블을 만들고 데이터를 나눔.
앨범, 영화, 책 테이블을 만든 후 앨범의 아티스트는 앨범에. 그리고 아이템에는 그것의 이름과 가격. pk, fk로 조인하여 조회할 수 있도록 설정함.
DTYPE은 앨범, 영화, 책 중 어떤 것을 조회할 것인가를 정해주는 컬럼

- 가장 정규화된 전략임. name, price가 중복이 안되고 앨범, 영화, 책 모두 각자 다른 데이터가 들어옴. jpa 전략과 가장 유사함.

조인 전략을 선택가능

```
package hellojpa;

import javax.persistence.*;

@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Item {
```

```
  @Id @GeneratedValue
  private Long id;
  private String name;
  private int price;
}
```

```
SELECT * FROM MOVIE ;

SELECT * FROM item ;
SELECT * FROM book ;
SELECT * FROM album ;
```

SELECT * FROM MOVIE ;

ACTOR	DIRECTOR	ID
bbb	aaa	1

(1 row, 3 ms)

SELECT * FROM item ;

ID	NAME	PRICE
1	cccc	10000

(1 row, 2 ms)

SELECT * FROM book ;

ISBN	AUTHOR	ID

(행 없음, 0 ms)

위 코드대로 작성하면 위 그림과 똑같이 매핑된다.
@Inheritance을 이용하여 joined 전략을 주었고, 이에 따라 각 객체들이 item의 pk를 이용하여 조인된다.

아래는 조인 전략을 써먹는 예시와 그 결과이다.

```
package hellojpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import java.util.ArrayList;
import java.util.List;

public class JpaMain {
  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello");

    EntityManager em = emf.createEntityManager();

    EntityTransaction tx = em.getTransaction();
    tx.begin();
    try {
      Movie movie = new Movie();
      movie.setDirector("aaa");
      movie.setActor("bbb");
      movie.setName("cccc");
      movie.setPrice(10000);
      em.persist(movie);

      em.flush(); // 영속성 컨텍스트에 있는 것을 db에 날리고
      em.clear(); // 영속성 컨텍스트에 있는 것을 모두 삭제하니까

      Movie findMovie = em.find(Movie.class, movie.getId()); // 여기서는 db에서 조회할 수 밖에 없다!
      System.out.println("findMovie.getName() = " + findMovie.getName());
```

```
        tx.commit();

    } catch (Exception e) {
        tx.rollback();
    } finally {
        em.close();
    }

    emf.close();

  }

}
```

```
Hibernate:
    select
        movie0_.id as id1_2_0_,
        movie0_1_.name as name2_2_0_,
        movie0_1_.price as price3_2_0_,
        movie0_.actor as actor1_7_0_,
        movie0_.director as director2_7_0_
    from
        Movie movie0_
    inner join
        Item movie0_1_
            on movie0_.id=movie0_1_.id
    where
        movie0_.id=?
findMovie.getName() = cccc
```

item 엔티티에 @DiscriminatorColumn(name = “DTYPE”) 추가시..

```
SELECT * FROM ITEM
```

SELECT * FROM ITEM;

DTYPE	ID	NAME	PRICE
Movie	1	cccc	10000

(1 row, 3 ms)

편집

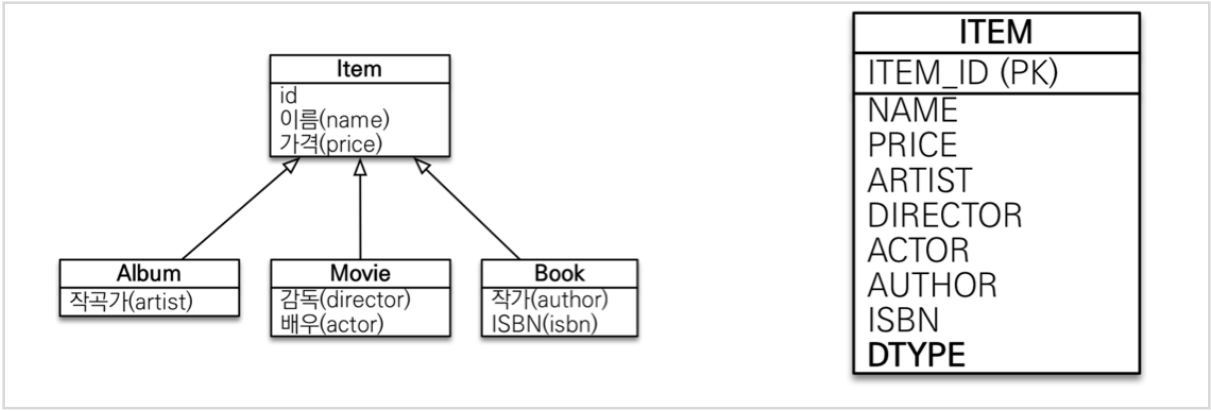
➡ 부모 엔티티에는 @DiscriminatorColumn, 자식 엔티티에는 @DiscriminatorValue.

➡ 필수는 아니고 운영상 있는게 좋음.

- 장점
  - 정규화가 되어있어서 데이터를 볼 때 제약조건 같은 걸 부모 클래스에 걸어서 보면 됨.
  - 외래키 참조 무결성 제약조건 활용 가능
    - item_id를 이용하면 item만 보면 됨!
  - 조인 전략을 정석으로 생각하자!

- 단점
  - 사실 큰 단점은 없음. 단일 테이블과 비교했을 때 복잡함 정도가 단점!
  - 조인을 잘 맞추면 성능이 저하되지도 않음.. 오히려 잘 나올수도?

단일 테이블 전략



그냥 한 테이블에 다 때려박아버림. DTYPE으로 어떤 카테고리를 정할 지를 결정함.
디비는 이렇게 돼있어도 설계 자체는 객체지향적..

```
package hellojpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import java.util.ArrayList;
import java.util.List;

public class JpaMain {
  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello");

    EntityManager em = emf.createEntityManager();

    EntityTransaction tx = em.getTransaction();
    tx.begin();
    try {
      tx.commit();
    } catch (Exception e) {
      tx.rollback();
    } finally {
        em.close();
    }

    emf.close();
```

```
  }

}
```

단일 테이블 전략은 DTYPE이 디폴트로 생성된다!
그냥 코드를 유지하고 생성해도 상관없지만, Inheritance 어노테이션에서 SINGLE_TABLE을 설정해주어도 된다.

```
package hellojpa;

import javax.persistence.*;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn
public class Item {

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public String getName() {
    return name;
  }

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

  public int getPrice() {
    return price;
```

```
    }

    public void setPrice(int price) {
      this.price = price;
    }

    @Id @GeneratedValue
    private Long id;
    private String name;
    private int price;
}
```

```
SELECT * FROM ITEM
```

SELECT * FROM ITEM;

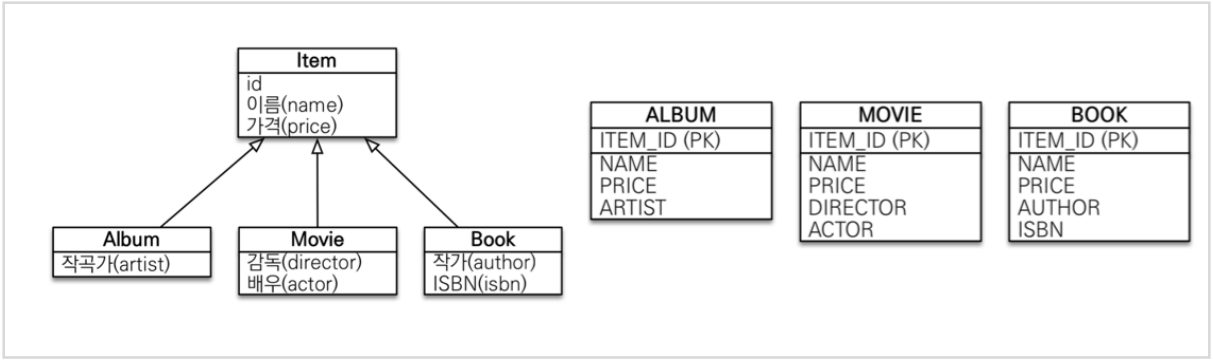
DTYPE	ID	NAME	PRICE	ARTIST	ISBN	AUTHOR	ACTOR	DIRECTOR
Movie!!!	1	cccc	10000	*null*	*null*	*null*	bbb	aaa

(1 row, 2 ms)

편집

- 장점 : 빠르다
- 단점
  - 자식 엔티티가 매핑한 컬럼은 모두 nullable로 들어가야한다..

구현 클래스마다 테이블 설계



말 그대로임.

Item, Album, Movie, Book 테이블을 모두 만듦.

```
package hellojpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import java.util.ArrayList;
import java.util.List;

public class JpaMain {
  public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("hello");

    EntityManager em = emf.createEntityManager();

    EntityTransaction tx = em.getTransaction();
    tx.begin();
    try {
      Item item = new Item();
      em.persist(item);

      Album album = new Album();
      em.persist(album);

      Movie movie = new Movie();
      em.persist(movie);
```

```
        Book book = new Book();
        em.persist(book);

        tx.commit();
    } catch (Exception e) {
        tx.rollback();
    } finally {
        em.close();
    }

    emf.close();

  }

}
```

```
Hibernate:
    call next value for hibernate_sequence
Hibernate:
    /* insert hellojpa.Item
        */ insert
        into
            Item
            (name, price, DTYPE, id)
        values
            (?, ?, 'Item', ?)
Hibernate:
    /* insert hellojpa.Album
        */ insert
        into
            Item
            (name, price, artist, DTYPE, id)
        values
            (?, ?, ?, 'Album', ?)
Hibernate:
    /* insert hellojpa.Movie
        */ insert
        into
            Item
            (name, price, actor, director, DTYPE, id)
        values
            (?, ?, ?, ?, 'Movie', ?)
Hibernate:
    /* insert hellojpa.Book
        */ insert
        into
            Item
            (name, price, ISBN, author, DTYPE, id)
        values
            (?, ?, ?, ?, 'Book', ?)
```

@Inheritance의 TABLE*PER*CLASS를 이용해준다.
item 클래스에 있는 속성인 name이나 Price값이 각자 클래스에 모두 중복돼서 들어가도록 함.
➜ 만약 item클래스를 이용해서 찾는다고 치면 모든 테이블 다 둘러봐야함.
=> 디비 설계자와 ORM(Object Relational Mapping임. 외우자) 전문가 둘 다 추천안함.
➜ 새로운게 추가가 될 때 마다 코드를 계속 짜야함. 변경 관점에서 정말 안좋음.

=> 정리!!
RDB는 상속관계가 없으나, 슈퍼타입 서브타입이라는 논리적 모델링 기법을 사용하여 상속을 할 수 있다.
이것을 물리적으로 실현시키는 방법이 세개인데, **구현 클래스마다 테이블 생성, 조인 전략, 단일 테이블 전략**이다.

<조인 전략과 단일 테이블 전략 둘 중에 하나를 써야한다면 뭘 쓸것인가?>
➜ 기본은 조인 전략인데, 너무 단순하면 단일 테이블을 쓰자. 비즈니스적으로 중요하고 복잡하면 조인 전략.

#JPA