Hanbit eBook

Realtime 83

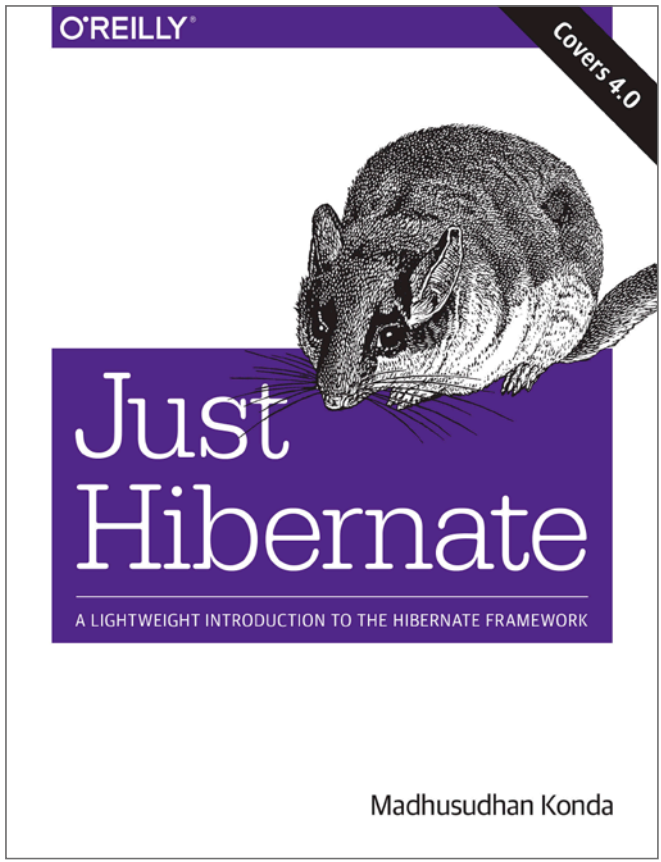
일주일 만에 끝내는

하이버네이트

Just Hibernate

마드후수단 콘다 지음 / 송기용 옮김

O'REILLY® 한빛미디어 Hanbit Media, Inc.



이 도서는 O'REILLY의
Just Hibernate
번역서입니다.

일주일 만에 끝내는 **하이버네이트**

초판발행 2014년 11월 05일

지은이 마드후수단 콘다 / **옮긴이** 송기용 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-722-4 15000 / **정가** 12,000원

총괄 배용석 / **책임편집** 김창수 / **기획·편집** 정지연
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 최송실
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 김옥현

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

저자_ **마드후수단 콘다**Madhusudhan Konda

마드후수단 콘다는 런던의 투자은행과 금융기관에서 경력직 자바 컨설턴트로 일하고 있다. 배포, 멀티 스레드, 확장 가능한 N-티어 아키텍처에 관심이 있으며, 금융 관련 고빈도high-frequency와 저지연low-latency 애플리케이션 아키텍트다. 집필에 남다른 애정이 있으며 멘토링에도 관심이 많다.

역자 소개

역자_ **송기용**

오픈소스 프레임워크를 사용한 웹 애플리케이션을 개발하고 있으며, 최근에는 함수형 프로그래밍에 관심이 많다. 현재 온라인 음원 서비스 회사에 재직 중이다.

저자의 말

저의 책들에 대한 긍정적이거나 부정적인 피드백, 리뷰, 의견, 제안, 글을 더 쓸 수 있는 힘이 되는 칭찬도 받았습니다. 저는 이 모두를 건설적으로 받아들였습니다. 여러분의 기대에 미치지 못한 글일지라도 지적해 주신다면 다음에는 최선을 다할 것입니다. "좋은 사람은 포기하지 않아요, 아빠."라고 아들 조슈아가 말한 것처럼 포기하지 않을 겁니다. 제가 즐겁게 글을 쓴 만큼 이 책을 즐기셨으면 좋겠습니다.

저나 제 글이 마음에 들지 않더라도 연락 주세요. 여러분의 의견은 항상 제게 큰 힘이 됩니다. 런던이나 주변에 계신다면 메시지나 메일을 주세요. 커피나 케이크를 함께 할 수 있을지도 모릅니다.

다음 Just 시리즈로 『Just Java 8』이 나올 예정입니다. 관심을 가져 주세요.

마드후수단 콘다

www.madhusudhan.com
m.konda@outlook.com
@mkonda007

역자 서문

처음 이 책의 번역을 맡았을 때 경험이 없어 생기는 걱정보다는 기쁜 마음이 더 컸습니다. 동료들과 IT 관련 기술이나 정보를 나누면서 느꼈던 즐거움을 좀 더 많은 분과 나눌 수 있지 않을까 하는 기대가 컸기 때문입니다.

하이버네이트나 JPA 관련 서적은 다른 기술 서적보다 찾기가 어려워 용어 선택이나 자연스럽지 못한 표현들 때문에 고민을 많이 했습니다.

이 책은 하이버네이트의 기초와 실무에서 사용할 수 있는 내용을 모두 담고 있습니다. 구글링을 통해 필요한 지식을 단편적으로 얻는 방법도 있지만, 책을 통해 하이버네이트 프레임워크의 시작부터 중고급 주제까지 살펴보는 것도 좋은 방법의 하나라고 생각합니다. 저 역시 실무에서 JPA 명세로는 한계가 있었으니까요.

이 책을 통해 개발자 여러분이 하이버네이트를 올바르게 사용하고자 하는 마음이 생긴다면 더할 나위 없이 기쁠 것 같습니다. 항상 독자에게 도움이 되는 책을 만들기 위해 개발자 입장에서 고민하시는 김창수 팀장님과 초보 역자인 저에게 하나부터 열까지 맞춰주느라 애써주신 정지연 과장님께 감사드립니다.

그리고 마지막으로 번역의 즐거움과 보람을 알게 해 준 아내 민정과 언제나 응원해 주시는 부모님께 감사의 마음을 전합니다.

일러두기

이 책의 대상 독자

이 책은 하이버네이트의 기초부터 중급까지 다룹니다. 예제 중심의 빠르고 쉽고 간단한 하이버네이트 입문서를 찾고 있다면, 이 책이 바로 그 책입니다. 하이버네이트 관련 프로젝트를 시작하기 전에 이 프레임워크를 배울 시간이 일주일밖에 없다면, 제대로 고르신 겁니다. 또한, 하이버네이트를 알고 있지만 세부 내용에 자신이 없다면 이 책을 통해 자신감을 얻을 수 있습니다.

이 책은 기술을 빠르게 습득하고자 하는 사람들을 위한 책이므로 하이버네이트에 능숙한 개발자에게는 적합하지 않습니다. 그러나 읽다 보면 흥미로운 부분을 찾을 수도 있으니 한번 훑어봐도 무방합니다.

다만 한 가지, 이 책은 하이버네이트 프레임워크에 대한 바이블이 아닙니다. 하이버네이트에 대한 내용을 적은 분량 안에 집약해서 넣었으므로 고급 기술의 책을 찾고 있다면 이 책을 추천하지 않습니다. 또한, 난관에 빠진 프로젝트의 솔루션을 찾으려고 이 책을 선택했다면 해결책을 찾기 힘들지도 모릅니다.

이 책을 집필한 이유

두 코스로 제공되는 식사처럼 배움에도 두 단계가 있다고 생각합니다. 첫 번째 코스는 간단하고 쉽지만 입맛을 돋우며 두 번째 코스에 대해 기대를 하게 만듭니다. 또한, 허기를 조금 채워줄 뿐만 아니라 두 번째 코스로 무엇이 나올지 맛보기를 제공합니다.

많은 책은 두 코스의 음식을 한꺼번에 제공합니다. 이러한 방식이 좋을 수도 있지

만, 사람들 대부분은 깊이 있는 기술을 배울만한 열정이나 시간 또는 공간이 부족합니다. 또한, 다른 레스토랑에서 파는 동일한 메뉴들이 우리를 늘 혼란스럽게 합니다. 메뉴가 계속 바뀌는 건 말할 것도 없고요.

첫 번째 코스는 손님에게 다음 코스를 위해 식당에 계속 머물러야겠다는 확신을 줄 수 있는, 흥미롭고 식욕을 돋우는 간단한 코스여야 한다고 생각합니다. 이런 코스를 제공하는 것은 어렵고 때로는 벅찬 일입니다.

일곱 살 난 아들 조슈아의 숙제를 함께하면서 기본과 원리를 쉽고 간단한 방법으로 가르치는 것이 얼마나 중요하고 어려운지 새삼 깨달았습니다. 아들의 수준에 맞게 쉬운 용어와 예제로 설명하면 아이는 아무리 어려운 주제라도 쉽게 이해했습니다.

기본에서 헤매는 많은 프로그래머와 개발자를 보았습니다. 이들은 동료에게 도움을 청하는 걸 때로는 부끄럽게 생각합니다. 그리고 신기술을 훈련하거나 학습할 시간 없이 정해진 단기간 내에 결과를 만들어야 하는 현실에 등 떠밀려 프로젝트를 진행하는 몇몇 사람도 본 적이 있습니다.

신기술을 배우는 데 관심은 있지만, 방대한 문서나 매뉴얼 때문에 배움을 미뤄두는 사람들도 있습니다. 이런 사람들은 열정적이고 새로운 업무를 바로 시작하고 싶어 하지만 많은 양의 책을 읽고 이해할 만한 인내심이나 시간이 없고, 간단한 책을 읽고 기술을 익혀서 실전에 바로 적용하기를 원합니다. 일단 그 기술을 이해하고 나면 좀 더 깊이 있는 지식을 원하게 됩니다.

저는 새로운 무언가를 배울 때 기본 코드를 실행해 보고 좀 더 나은 코드를 따라 해 보기도 합니다. 즉, 하이버네이트 프레임워크가 어떻게 동작하는지 이해하기 시작

하면 다른 경로를 통해 배움의 갈증을 해소합니다. 이때부터 난도가 높고 깊이 있는 매뉴얼과 명세서 그리고 두꺼운 책을 찾기 시작합니다.

Just 시리즈는 독자들이 간단하더라도 재미있고 빠르게 읽을 수 있도록 썼습니다. 기술을 실용적인 예제 중심으로 핵심을 집어내서 전달하려 했습니다. 물론 쉽게 썼고요. Just 시리즈는 여러분에게 충분한 지식과 실무 프로젝트에서 일을 시작할 수 있는 자신감을 줄 것입니다.

이 책의 구성

이 책은 총 8장으로 구성되어 있습니다. 각 장은 한 개 또는 두 개의 특정 주제를 다루며, 소스 코드를 제공합니다.

각 장의 내용은 다음과 같습니다.

1장 기초

하이버네이트 사용 환경을 설정합니다. 문제 범위를 정의하고, JDBC를 이용한 해결법을 연습해 보고, 하이버네이트를 도입해서 문제점을 개선하겠습니다. 여기서 하이버네이트를 살짝 맛보게 됩니다.

2장 기본 개념

하이버네이트가 해결하려는 문제에 대해 알아봅니다. 하이버네이트 프레임워크의 실행과 동작을 살펴보기 위해 프레임워크에 대해 좀 더 깊이 있게, 하이버네이트의 핵심 부분을 자세히 알아보겠습니다.

3장 어노테이션

어노테이션을 이용하여서 하이버네이트 애플리케이션을 생성하는 것에 집중할 것입니다. 또한, 뒷장에서 다룰 하이버네이트 어노테이션에 대한 준비 작업을 위해 어노테이션의 기본을 다지게 됩니다.

4장 콜렉션 영속화

영속화된 콜렉션은 개발자에게 어려운 과제입니다. 이 장은 콜렉션을 가지고 동작하는 방법과 영속성의 메커니즘을 이해하는 데 도움을 주기 위해 할애하였습니다.

5장 연관 관계

하이버네이트에서 제공하는 연관 관계와 관계를 알아봅니다. '일대일', '다대다' 같은 기본 연관 관계에 관련된 예제들을 다룹니다. 이 장은 주제에서는 살짝 벗어나지만 가능하면 간단하게 설명하려고 노력했습니다. 매우 중요한 장으로, 연관 관계를 제대로 이해하는 것만으로도 반은 성공입니다.

6장 고급 개념

캐싱, 상속 전략, 타입, 필터와 같은 고급 주제를 다룹니다. 프레임워크와 프레임워크가 상속, 캐싱, 그 밖의 기능과 관련하여 제공하는 것에 대해 좀 더 깊이 알 수 있을 것입니다.

7장 하이버네이트 질의어

하이버네이트는 SQL과 유사하게 자체 질의어가 있습니다. 하이버네이트 질의어(HQL)에 대해 소개하고 예제와 함께 API에 대해 알아봅니다.

8장 자바 퍼시스턴스 API

하이버네이트 관점에서 자바의 영속성 세계의 표준인 자바 퍼시스턴스 API(JPA)에 대해 살펴봅니다. 또한, JPA 표준 구현을 위한 하이버네이트의 지원과 애플리케이션에서 JPA 표준을 어떻게 활용할 수 있는지 살펴보겠습니다.

참고사항

이 책에서 사용하는 소스 코드는 https://github.com/madhusudhankonda/jh.git에서 내려받을 수 있습니다.

이 아이콘은 제안이나 팁을 의미한다.

이 아이콘은 일반적인 주석을 의미한다.

이 아이콘은 경고나 주의를 의미한다.

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. **eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.**

   500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. **무료로 업데이트되는, 전자책 전용 서비스입니다.**

   종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

1 | 기초

두 개의 서로 다른 소프트웨어 세계가 있는데, 하나는 객체로 알려진 자바 세계고 다른 하나는 데이터가 왕인 관계형 데이터베이스 세계다.

자바 개발자는 작업할 때 언제나 객체를 다룬다. 객체는 현실 세계를 모델링하는 상태state와 행동behavior을 나타낸다. 자바 애플리케이션에서 객체의 영속성object persistence은 필요조건이다. 상태는 내구성 있는 스토리지durable storage에 영속화되도록 모델링되어 있기 때문에 영구적이다.

하지만 데이터를 보관할 때는 관계형 데이터베이스에 의존하게 된다. 관계형 데이터베이스에서 데이터는 전통적으로 로우row와 컬럼column 형태로 데이터 간의 관계relationship나 연관 관계association를 나타낸다. 자바 개발자에게 자바 객체를 관계형 세상으로 가져온다는 것은 항상 어렵고 복잡한 일이다. 이런 과정을 객체-관계 연결ORM, Object-Relational Mapping이라 한다.

이 장에서는 객체 영속성 문제를 살펴봄으로써 하이버네이트에 대한 논의를 시작하겠다. 이런 과제에 직면한 우리를 도와줄 JDBC와 하이버네이트Hibernate 같은 여러 기술과 툴에 대해 알아보고, 이 기술들을 비교 대조해 볼 것이다. 그리고 하이버네이트에서 어떻게 쉽고 편하게 영속화된 객체 관계형 모델을 얻는지 살펴본다.

1.1 하이버네이트의 탄생

온라인 뱅킹 시스템을 설계한다고 생각해 보자. 은행이 계좌 정보, 개인 정보, 거래 내역 등의 데이터를 안전하게 보관할 거라고 기대할 것이다. 이는 애플리케이션 정보가 오랫동안 파손되지 않고 영구적인 저장 공간에 있어야 한다는 걸 의미한다. 뱅킹 애플리케이션의 고객, 주소, 계정, 기타 도메인 객체는 설계대로 영속화된 데

이터로 사용될 것이다. 이 애플리케이션을 통해 영속화된 데이터들은 애플리케이션보다 더 오래 남아 있을 것이다. 다시 말해, 온라인 뱅킹을 하다가 폰뱅킹으로 바꾼다 하더라도 뱅킹 애플리케이션에서 생성된 데이터들은 원할 때 볼 수 있고 사용할 수 있어야 한다.

이제 영속화된 객체(객체의 상태는 영속화할 데이터)는 현실 세계 대부분의 애플리케이션을 위한 필요조건이라는 것을 알게 되었다. 그리고 데이터를 저장하기 위해서 데이터베이스라는 내구성 있는 저장 공간이 필요하다. 다양한 부가기능을 제공하는 데이터베이스 벤더들(오라클, MySQL, DB2, JavaDB 등)은 많다.

어떻게 하면 객체 간 관계를 데이터베이스로 영속화할 수 있을까?

기업에서는 프로그래밍 플랫폼으로 객체지향 언어(Java 등)를, 데이터베이스로는 관계형 데이터베이스(Oracle, MySQL, Sybase 등)를 도입하고 있다. 소위 객체 관계형 임피던스object-relational impedance 불일치에도 불구하고 이 두 소프트웨어 기술은 현실 세계 대부분의 애플리케이션에 꼭 필요하다.

다음 장에서는 이 불일치에 대해 자세하게 살펴보고 여기서는 주요 사항을 다음과 같이 간단히 소개한다.

- 상속은 객체지향 프로그래밍의 기본 원리다(상속 개념 없이는 객체 연관 관계를 설계할 수 없으며, 데이터베이스는 상속 개념을 알지 못한다).
- 일대일, 일대다, 다대다와 같은 다양한 객체 관계의 구현을 해도, 데이터베이스에서는 모든 관계를 지원하지 않으므로 상속은 결국 실패로 끝난다.
- 마지막으로 동일성identity의 불일치가 있다. 객체는 동일성과 동등성equality 둘 다를 가지고 있지만, 데이터베이스 레코드는 컬럼 값으로만 식별된다.

개발자는 직접 작성한 프레임워크, 기술적 솔루션, 전략을 통해 객체-관계 간 차이에서 오는 불편함을 줄일 수 있다.

자바에는 데이터베이스 접근을 위한 표준 도구가 있는데, 이를 JDBC Java Database Connectivity API Application Programming Interface라고 한다. 최근까지도 JDBC API는 자바 애플리케이션에서 잘 활용되고 있다. 이 API는 작은 규모의 프로젝트에 적합하지만, 도메인 모델이 복잡해질수록 프로젝트는 매우 비효율적으로 변한다(그리고 때때로 통제가 불가능하다). 또한, 반복되는 많은 코드와 노동력이 필요할 뿐만 아니라 객체-관계 모델 매핑을 다루는 것 역시 필요 이상의 큰 비용이 필요하다.

이는 개발자에게 고통스러운 부분이다. 큰 문제 없이 데이터를 영속화할 수 있는 간단한 도구가 생기길 모두가 기다렸다. 하이버네이트 팀은 ORM 매핑 공간에서 발생하는 차이점을 발견했고 개발자의 일을 좀 더 편하게 해주는 간단한 프레임워크를 만들게 되었다.

이것이 하이버네이트의 시작이다. 하이버네이트는 나오자마자 큰 성공을 거두었고 단순함과 강력함을 특징으로 단숨에 ORM 툴 분야에서 가장 인기 있는 오픈소스로 자리매김하였다.

1.2 문제 범위

하이버네이트에 대해 자세히 알아보기 전에 하이버네이트가 나오게 된 이유를 예를 통해 알아보자.

사람들은 영화 보는 것을 좋아한다. 하지만 영화가 개봉할 때마다 영화를 다 볼 수 있을 만큼 시간이 많지 않다. 그래서 보고 싶은 영화를 '위시 리스트'로 만든다. 그러다 어느 날 문득 'JustMovies'라는 애플리케이션을 만들기로 마음먹는다. 'JustMovies'는 웹 기반 애플리케이션으로, 사용자는 개인 계정을 만들고 자신만의 영화 위시 리스트를 작성할 수 있다. 또한, 언제든지 웹사이트에서 위시 리스트에 영화를 추가하거나 수정 또는 삭제할 수 있다.

이때, 사용자마다 위시 리스트를 저장해야 하므로 반드시 데이터베이스 같은 내구성 있는 스토리지에 각 리스트가 저장되어야 한다. 자, 우선 데이터베이스에 저장하고 조회해서 가져오는 간단한 자바 애플리케이션을 만들어보자.

1.2.1 MovieManager 애플리케이션

데이터베이스에 영화 정보를 저장하고, 조회하고, 찾아서 가져오는 일을 주로 하는 자바 애플리케이션인 'MovieManager'가 있다. 자바로 구현된 애플리케이션이고 영화 정보를 저장해야 하므로 데이터베이스 테이블이 필요하다. 영화와 관련된 데이터는 [표 1-1]과 같이 MOVIES 테이블에 로우 형태로 저장한다.

[표 1-1] MOVIES 테이블

ID	TITLE	DIRECTOR	SYNOPSIS
1	Top Gun	Tony Scott	Maverick is a hot pilot…
2	Jaws	Steven Spielberg	A tale of a white shark!

각 컬럼은 `VanillaMovieManager` 애플리케이션에서 `Movie` 인스턴스를 나타낸다. 하이버네이트 없는 세상에 살고 있다고 가정해 보자. 필요한 것들을 구현할 수 있기를 바라면서 JDBC를 사용하여 몇 줄의 예제 코드를 작성하게 될 것이다.

JDBC 사용하기

어떤 데이터베이스 애플리케이션이든 시작은 데이터베이스를 연결하고 유지하는 일이다. 커넥션connection은 데이터베이스와 연결된 게이트웨이다. 자바 애플리케이션에서 데이터 작업을 수행하는 JDBC는 데이터베이스 속성에 기반을 둔 커넥션을 생성하는 커넥션 API를 제공한다. 데이터베이스 제공자는 일반적으로 데이터베이스 연결 메커니즘을 가지고 있는 클래스를 구현하고 있다. 예를 들어, MySQL에서는 `com.mysql.jdbc.Driver`이고, 더비derby(JavaDB)에서는 `org.apache.derby.jdbc.EmbeddedDriver`다.

이 책에서는 MySQL을 다룬다. 프로젝트와 데이터베이스 설정에 대한 자세한 설명은 1.7 하이버네이트 설정하기를 참고하기 바란다.

다음 소스 코드의 createConnection은 데이터베이스 연결을 생성하는 과정을 보여준다. 이 부분에서 드라이버 클래스를 초기화하고 DriverManager를 이용하여 커넥션을 얻는다.

```
public class VanillaMovieManager {
     private Connection connection = null;

     //데이터베이스 속성
     private String url = "jdbc:mysql://localhost:3307/JH";
     private String driverClass = "com.mysql.jdbc.Driver";
     private String username = "mkonda";
     private String password = "mypass";
     …
     private Connection getConnection() {
         try {
             Class.forName(driverClass).newInstance();
             connection = DriverManager.getConnection(url, username, password);
         } catch (Exception ex) {
             System.err.println("Exception:"+ ex.getMessage());
         }
         return connection;
     }
}
```

데이터베이스로 연결되고 유지되면, 다음 단계로 영속화를 위한 메소드를 작성하고 movie 엔티티Entity를 조회한다. JDBC를 다뤄봤다면 작성된 메소드 코드가 익숙할 것이다.

그다음, 영화 정보를 데이터베이스에 저장하는 메소드와 조회해서 가져오는 메소드를 추가한다. 이 메소드를 각각 persistMovie와 queryMovies라 하자. 다음 코드는 이들 메소드의 구현부다.

```
public class VanillaMovieManager {
     private String insertSql = "INSERT INTO MOVIES VALUES (?,?,?,?)";
     private String selectSql = "SELECT * FROM MOVIES";
     …

     private void persistMovie() {
         try {
             PreparedStatement pst = getConnection().prepareStatement(insertSql);
             pst.setInt(1, 1001);
             pst.setString(2, "Top Gun");
             pst.setString(3, "Action Film");
             pst.setString(4, "Tony Scott");

         // statement 실행하기
             pst.execute();
             System.out.println("Movie persisted successfully!");

         } catch (SQLException ex) {
             System.err.println(ex.getMessage());
             ex.printStackTrace();
         }
     }
     private void queryMovies() {
         try {
             Statement st = getConnection().createStatement();
             ResultSet rs = st.executeQuery("SELECT * FROM MOVIES");
             while (rs.next()) {
                 System.out.println("Movie Found: "
                           + rs.getInt("ID")
```

```
                        + ", Title:"
                        + rs.getString("TITLE"));
            }
        } catch (SQLException ex) {
            System.err.println(ex.getMessage());
        }
    }
}
```

예제 코드의 내용은 다음과 같다.

1. 커넥션과 `insertSql` 문자열 변수를 통해 `PreparedStatement`를 생성한다.
2. `Preparedstatement` 객체에 영화 정보(컬럼 번호에 맞는 컬럼 값: 1은 ID, 2는 제목 등)를 넣는다.
3. `PreparedStatement` 객체를 실행하면 `Movies` 테이블에 영화 정보가 추가된다.
4. 데이터베이스의 모든 영화를 대상으로 조회 쿼리를 실행하고, 콘솔에 리스트를 출력한다.

각 단계는 추가 설명 없이도 쉽게 이해될 것이다. `PreparedStatement`를 생성하고, 각 컬럼에 맞는 데이터를 넣은 후 실행한다. 실행이 잘 되었는지 확인하려면 모든 영화 정보의 조회와 콘솔 출력을 위한 `SELECT` 문 쿼리를 실행하면 된다.

그러나 몇 가지 주의 사항이 있다.

- 컬럼 값을 입력(또는 조회)하기 위해 미리 정의된 SQL문을 사용한다.
- 컬럼 값은 컬럼 순번(또는 컬럼 이름)으로 하나씩 설정한다.
- SQL문이 제대로 실행되지 않으면 `SQLException`이 발생한다.

간단한 프로그램에서는 값과 함께 코드를 생성하고 실행하는 방법으로도 충분하다. 그러나 현실 세계의 프로그램은 훨씬 더 복잡하다. 비즈니스 로직과 상관없는

엄청난 양의 코드를 기꺼이 기쁜 마음으로 작성하고 관리할 수 있다면 JDBC를 사용해도 된다. 객체 간 복잡한 관계나 많은 테이블을 다뤄야 한다면 JDBC 도입은 문제가 된다. 이렇게 데이터를 다루는 것은 객체지향 원리를 사용하는 어떠한 종류나 형태에도 해당하지 않는다.

1.2.2 Movie 애플리케이션 구현

Movie 객체를 바로 영속화하기 위해 유틸리티 클래스에서 제공하는 persist() 같은 메소드를 호출하면 좋지 않을까? 객체지향 프로그래머가 이런 편리한 기능을 바라는 것이 죄가 되겠는가!

이를 위해 영화 한 편을 나타내는 POJO[Plain Old Java Object]를 만들겠다. 개봉했거나 아직 개봉하지 않은 모든 필름 영화를 위해 새로운 Movie 객체를 생성한다. 다음은 Movie POJO의 선언부다.

```
public class Movie {
     private int id = 0;
     private String title = null;
     private String synopsis = null;
     private String director = null;
     …
     // Setters와 getters 생략
}
```

따라서 이제 POJO 객체를 데이터베이스 테이블 MOVIES로 영속화하는 기능이 필요하다. 본질적으로 객체 모델(Movie 객체)을 관계형 모델(테이블의 컬럼)로 변환하는 것이다.

변환 작업을 위해 MoviePersistor 클래스를 만들어보자.

```
// Pseudocode
public class MoviePersistor {
     public void persist(Movie movie) {
        // 이 부분에서 Movie 객체가 영속된다
     }
     public void fetch(String title) {
        // 이 부분에서 title을 매개변수로 movie 객체를 가져온다
     }
     …
}
```

아직 persist와 fetch 기능을 구현하지 않았다(이것은 프로그램의 주제가 아니다). 자, MoviePersistor 유틸리티 클래스를 이용하여 Movie를 영속화하는 것이 얼마나 쉬운지 살펴보자.

```
//Pseudocode
MoviePersistor moviePersistor = new MoviePersistor();
Movie movie = new Movie();
movie.setId(1);
movie.setTitle("Jaws");
movie.setDirector("Steven Spielberg");
movie.setSynopsis("Story of a great white shark!");

moviePersistor.persist(movie);
```

멋지지 않은가? 한 편의 필름 영화에 해당하는 하나의 POJO는 유틸리티 클래스를 통해 객체 모델에서 관계형 모델로, 데이터베이스 테이블 하나의 레코드로 영속화된다.

Persist와 fetch 메소드를 실제로 구현하지 않을 것을 제외하고는 모든 것이 좋다. 이 기능을 구현하려면 데이터베이스 연결 기능뿐만 아니라 객체를 로우로 변환

하는 메커니즘도 필요하다(객체 속성을 데이터베이스 컬럼으로 매핑한 것과 같다).

이러한 변환의 핵심 부분과 영속화 메커니즘을 감추는 클래스를 통해 자신만의 프레임워크를 만들 수 있다(화면 뒤에 오래된 JDBC 구문을 숨기는 데 좋은 방법이 될 것이다). 비록 이 프레임워크를 구축하는 것이 복잡한 일은 아니지만, 구현하는 데 많은 시간과 노력이 필요하다.

시간이 지나면 조직의 영속성 관련 요구사항이 변하거나 심지어 데이터베이스를 오라클에서 MySQL로 이관해야 할 수도 있다. 이는 프레임워크가 아주 포괄적이고 기능적이며 기술적인 요구사항에 대해 많은 부분을 제공해야 한다는 의미이기도 하다.

필자의 경험에 비춰보면 개인적인 프레임워크는 다루기 힘들고 유연성과 확장성이 부족하며 때로는 구닥다리가 될 수도 있다. 조직이 원하는 요구사항이 말도 안 되는 것이 아니라면(회사에서 화성에 데이터를 남기길 원할지도 모른다) 인터넷 검색으로 여기서 언급한 사항을 가장 만족하는 프레임워크를 선택할 것을 추천한다.

그런데 이미 관계형 데이터베이스로 객체를 영속화하는 기능을 가진 훌륭한 프레임워크가 존재한다. 그것이 바로 하이버네이트다!

자 그럼, 하이버네이트로 기존의 동일 메소드를 어떻게 리팩토링하는지 살펴보자.

```
public class BasicMovieManager {
     private void persistMovie(Movie movie) {
         Session session = sessionFactory.getCurrentSession();
         …
         session.save(movie);
     }
     …
}
```

한 줄의 코드 session.save(movie)의 실행으로 Movie 인스턴스를 데이터베이스로 저장하는 것을 보았는가? 앞에서 원했던 것이 아닌가? 객체지향 방식으로 간단히 영속화 객체를 저장하는 클래스인 Hibernate API 클래스는 쉽고 편하게 자바 객체를 다루기 위한 메소드들을 제공한다. 이는 곧, 다량의 카페인을 섭취하고 고심하면서 프레임워크를 작성하거나 JDBC를 사용해서 많은 양의 코드를 작성하지 않아도 된다는 의미다.

하이버네이트는 객체 영속화 기능을 제공하지만, 일회성 설정과 사용 목적에 맞는 매핑 정보는 직접 제공해야 한다. 다음 절에서 이 내용을 자세히 살펴보겠다.

1.3 하이버네이트 사용하기

하이버네이트 애플리케이션을 생성하기 위한 일반적인 단계는 다음과 같다.

1. 데이터베이스 연결 설정하기
2. 매핑 정보 생성하기
3. 클래스 영속화하기

다음은 MovieManager 애플리케이션을 자바-하이버네이트 버전으로 개발하는 일반적인 단계다.

1. Movie 도메인Domain 객체 생성하기(데이터 테이블에 상응하는 도메인 모델 POJO)
2. 하이버네이트 속성properties 파일과 매핑 파일 같은 설정 파일 생성하기
3. Movie 객체를 관리(입력, 변경, 삭제, 조회)할 테스트 클라이언트 생성하기

앞에서 Movie POJO를 이미 준비했으므로 다시 만들 필요는 없다.

하이버네이트 애플리케이션의 핵심은 설정configuration에 있다. 두 개의 설정 파일이 필요한데, 하나는 데이터베이스 연결을 생성하고, 다른 하나는 객체와 테이블을 매

핑하는 역할을 한다. JDBC를 살펴보면 애플리케이션에 데이터베이스 정보를 제공하고, 데이터를 다루기 위해 커넥션을 연결한다. 매핑 설정은 테이블의 컬럼에 매핑된 객체 속성을 정의한다. 이번 장의 목표는 빨리 시작해 보는 것이므로 상세한 부분까지는 들어가지 않겠다.

다음 절에서 하이버네이트 애플리케이션을 만드는 과정을 자세히 살펴보자.

1.4 데이터베이스 연결 설정하기

데이터베이스를 연결하려면 하이버네이트에 데이터베이스, 테이블, 클래스, 기타 장비에 대한 상세 정보를 제공해야 한다. 상세 정보는 XML 파일 형태(일반적인 형태는 hibernate.cfg.xml)로 제공하거나 '이름/값' 형태의 텍스트 파일(일반적인 형태는 hibernate.properties)로 제공한다. 여기서서는 XML 형태를 사용하겠다. 파일명은 hibernate.cfg.xml로 하고, 이 설정 파일은 하이버네이트에서 자동으로 로드된다.

다음은 xml 설정 파일에 대한 설명이다. MySQL 데이터베이스를 사용하므로 MySQL 데이터베이스를 위한 연결 상세 정보는 다음 hibernate.cfg.xml 파일에 선언되어 있다.

```
<hibernate-configuration>
    <session-factory>
        <property name="connection.url">
            jdbc:mysql://localhost:3307/JH
        </property>
        <property name="connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <property name="connection.username">
            mkonda
        </property>
```

```
        <property name="connection.password">
            password
        </property>
        <property name="dialect">
            org.hibernate.dialect.MySQL5Dialect
        </property>
        <mapping resource="Movie.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

이 파일은 MySQL 데이터베이스에 실시간 연결을 위한 정보를 갖고 있다.

앞선 속성들을 '이름/값' 형태로 표현할 수도 있다. 예를 들면, hibernate.properties 텍스트 파일에서 '이름/값' 형태와 같이 동일 정보를 나타낼 수 있다.

```
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost:3307/JH hibernate.dialect =
org.hibernate.dialect.MySQL5Dialect
```

connection.url은 어느 URL에 연결해야 할지를 가리키고, driver_class는 커넥션 생성에 맞는 Driver 클래스를 보여준다. 그리고 'dialect'에는 사용하는 데이터베이스의 dialect를 알려준다(여기서는 MySQL).

hibernate.properties 파일을 따른다면 모든 속성명은 hibernate.* properties와 같은 패턴으로 'hibernate' 접두사와 함께 정의해야 한다. 설정 파일을 제공하고 나면 매핑 파일과 설정 파일의 위치 정보도 제공해야 한다. 앞서 언급했듯이 매핑 파일은 객체 속성과 맞는 로우와 컬럼 값에 대한 객체 속성을 가지고 있다. 매핑 정보는 각 파일에 선언되었는데, 일반적으로 '.hbm.xml' 접미사를 사용한다. 다음과 같이 앞에서 본 설정 파일의 mapping 요소를 포함하는 매핑 설정 파일에 대한 정보를 제공해야 한다.

```
<hibernate-configuration>
    <session-factory>
        ...
        <mapping resource="Movie.hbm.xml" />
        <mapping resource="Account.hbm.xml" />
        <mapping resource="Trade.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

resource 속성은 하이버네이트에서 로드할 매핑 리소스명을 가리킨다. 예를 들어, Movie.hbm.xml은 Movie 테이블과 Movie 객체 간 매핑 방법에 대한 상세 정보를 가진 매핑 파일이다. Account.hbm.xml과 Trade.hbm.xml에서도 마찬가지로 매핑 정보를 볼 수 있다. 이 매핑 파일에 대해서 잠시 살펴보겠다.

하이버네이트는 설정 파일로 무엇을 할까?

하이버네이트 프레임워크는 SessionFactory를 생성하기 위해 설정 파일을 로드한다. SessionFactory는 Session 생성을 위한 Thread-safe한 전역 클래스다. 여기서는 하나의 SessionFactory를 생성하고 애플리케이션 전반에 걸쳐 공유하여 사용한다.

SessionFactory는 오직 하나의 데이터베이스를 위해 정의된 클래스라는 걸 기억하자. 예를 들어, MySQL에서 또 다른 데이터베이스를 사용한다고 하자. 해당 데이터베이스를 위한 별개의 SessionFactory를 생성하려면 hibernate.hbm.xml 파일에 적합한 설정을 정의해야 한다.

SessionFactory의 역할은 Session 객체를 생성하는 것이다. Session은 데이터베이스로 연결된 하나의 게이트웨이다. Session의 임무는 테이블에 레코드를 저장하고, 로딩하며, 조회해서 가져오는 데이터베이스의 모든 동작을 처리하는 것이다.

또한, 애플리케이션 간 트랜잭션을 유지하기도 한다. 데이터베이스 접속을 포함한 이런 동작은 '트랜잭션transaction'이라 불리는 하나의 처리 단위에 포함된다. 그래서 트랜잭션 안의 모든 작업은 성공적으로 완료되거나 롤백될 수 있다.

설정 파일은 SessionFactory 인스턴스를 통해 Session을 생성하기 위해 사용된다는 것을 기억하자. Session 객체는 Thread-safe하지 않으므로 다른 클래스 사이에서 공유되어서는 안 된다는 점도 기억하자. 앞으로 진행 과정을 통해 Session을 어떻게 사용해야 할지를 자세히 알아보겠다.

1.5 매핑 정의 만들기

데이터베이스 연결 설정이 준비되면, 다음 단계는 객체 테이블 간 매핑 정의가 된 Movie.hbm.xml 파일을 만드는 것이다. 다음 XML 코드는 MOVIES 테이블에 맞는 Movie 객체의 매핑 정보를 정의하고 있다.

```
<hibernate-mapping>
    <class name="com.madhusudhan.jh.domain.Movie" table="MOVIES">
        <id name="id" column="ID">
            <generator class="native"/>
        </id>
        <property name="title" column="TITLE"/>
        <property name="director" column="DIRECTOR"/>
        <property name="synopsis" column="SYNOPSIS"/>
    </class>
</hibernate-mapping>
```

이 매핑 파일에서 살펴볼 것이 많다. hibernate-mapping 요소는 객체-테이블 간 매핑 정보를 모두 가지고 있다. 객체 각각의 매핑 정보는 이 class 요소 하위에 선언되고, class 태그의 name 속성은 POJO 도메인 클래스 com.madhusudhan.

jh.domain.Movie를 참조한다. 하지만 table 속성은 객체를 영속화할 MOVIES 테이블을 참조한다.

나머지 속성은 객체 변수와 테이블 컬럼 간의 매핑 정보를 나타낸다(예를 들어, id는 ID, title은 TITLE, director는 DIRECTOR). 각 객체는 테이블의 기본키Primary Key와 같은 고유한 식별자를 가져야 한다. 네이티브native 전략을 사용한 id 태그를 구현함으로써 식별자를 설정할 수 있다. 아직 id 태그나 생성 전략generation strategy에 관심을 가질 필요는 없다. 이는 다음 장에서 자세히 살펴보겠다.

1.6 객체 영속화하기

지금까지 하이버네이트 설정을 알아보았다. 이제 하이버네이트로 객체를 영속화하는 클라이언트를 만들어보자.

Session 객체를 생성할 SessionFactory 인스턴스가 필요하다. 다음은 SessionFactory 클래스 생성을 위한 초기 설정 부분이다.

```
public class BasicMovieManager {
     private SessionFactory sessionFactory = null;

     // 하이버네이트 4.2 버전을 이용해서 SessionFactory 생성
     private void initSessionFactory(){
         Configuration config = new Configuration().configure();
         // Properties 설정파일을 가지고 Registry 객체 생성
         ServiceRegistry serviceRegistry = new ServiceRegistryBuilder().applyS
ettings(config.getProperties()).buildServiceRegistry();

         // SessionFactory 생성
         sessionFactory = config.buildSessionFactory(serviceRegistry);
     }
     …
}
```

하이버네이트를 시작할 때 클래스패스classpath에서 hibenate.cfg.xml이나 hibernate.properties와 같은 디폴트 파일명을 찾아서 로드하므로 속성과 설정, 매핑 파일을 꼭 명시할 필요는 없다.

SessionFactory 클래스 초기화를 위한 설정은 글을 쓰고 있을 당시 최신 버전인 4.2에 해당된다. 하이버네이트 4.x 버전부터 서비스 레지스트리service registries가 소개되었고, 이는 뒷부분에서 다루겠다. 3.x 대 버전에서는 Configuration 객체를 생성하기 위해 configure 메소드는 클래스패스 내 hibernate.cfg.xml(또는 hibernate.properties) 파일을 찾는다. 그리고 생성된 Configuration 객체는 SessionFactory 인스턴스를 만든다. 하이버네이트 4.x 이전 버전에서는 SessionFactory를 초기화하기 위해 다음 코드를 사용한다.

```
// 하이버네이트 3.x 버전을 이용해서 SessionFactory 생성
private void init3x(){
   sessionFactory =
       new Configuration().configure().buildSessionFactory();
}
```

4.x 버전에서는 Configuration 객체로부터 속성 매핑 정보를 얻는 서비스 레지스트리의 도입으로 약간 변경된 부분이 있다.어느 버전을 선택하더라도 이렇게 생성된 SessionFactory는 동일한 역할을 하고 Session 또한 마찬가지다.

1.6.1 Persist 메소드 만들기

이제 BasicMovieManager 클래스를 만들어보자. 영화 정보를 영속화하기 위해 Session의 save 메소드를 이용하여 클래스 내에는 persist 메소드를 선언한다.

```
public class BasicMovieManager {
     private void persistMovie(Movie movie) {
```

```
        Session session = sessionFactory.getCurrentSession();
        session.beginTransaction();
        session.save(movie);
        session.getTransaction().commit();
    }
}
```

간단해 보이지 않는가? 객체를 저장하는 비즈니스 기능에 집중하는 것 말고 불필요하거나 반복적인 코드는 전혀 작성할 필요가 없다.

SessionFactory로부터 Session을 얻는다. 그러고 나서 트랜잭션 객체를 생성하고(트랜잭션에 관해서는 다음 장에서 자세히 살펴볼 예정이다). Session.save 메소드를 통해 매개변수로 들어온 Movie 객체를 영속화한다. 마지막으로 트랜잭션 커밋Commit을 실행해서 데이터베이스에 Movie 객체를 영구적으로 저장한다.

1.6.2 영속화된 데이터 테스트

영속화된 데이터를 테스트하는 방법은 두 가지가 있다. 데이터베이스에서 네이티브 SQL 쿼리를 실행하는 방법과 테스트 클라이언트 프로그램을 만드는 방법이다.

SELECT * FROM MOVIES와 같은 SQL 쿼리를 데이터베이스에서 실행하면 애플리케이션을 통해 저장된 레코드 전체를 조회한다. select 쿼리문의 결과는 [표 1-2]와 같이 출력된다.

[표 1-2] MOVIES

ID	TITLE	DIRECTOR	SYNOPSIS
1	Top Gun	Tony Scott	Maverick is a hot pilot…
2	Jaws	Steven Spielberg	A tale of a white shark!

다른 방법은 테스트 클라이언트(findMovie라고 하자)의 새 메소드를 만드는 것이다.

이 메소드는 Session에서 제공하는 load 메소드를 이용하여서 데이터베이스의 레코드를 조회한다. 영화 정보를 조회하기 위해 영화 ID를 인수로 넘겨서 findMovie를 호출한다.

```
public class BasicMovieManager {
      …
      private void findMovie(int movieId) {
          Session session = sessionFactory.getCurrentSession();
          session.beginTransaction();

          Movie movie = (Movie)session.load(Movie.class, movieId);

          System.out.println("Movie:"+movie);
          session.getTransaction().commit();
      }
}
```

Session API에서 제공하는 load 메소드는 주어진 식별자로 해당하는 Movie 객체를 조회한다. 하이버네이트의 보이지 않는 부분에서 SELECT 문을 사용할 거라 생각한다면, 맞다!

테이블의 모든 영화 정보를 조회하고 싶다면, "from Movie"라는 간단한 쿼리문과 함께 Query 객체를 생성하고 실행하면 된다. Query 객체(session.createQuery로 생성된)의 list 메소드는 다음과 같이 영화 리스트를 반환한다.

```
public class BasicMovieManager {
      // 모든 영화 조회하기
      private void findAll() {
          Session session = sessionFactory.getCurrentSession();
          session.beginTransaction();
          List<Movie> movies = session.createQuery("from Movie").list();*
```

```
            session.getTransaction().commit();
            System.out.println("All Movies:"+movies);
        }
        …
}
```

1.7 하이버네이트 설정하기

하이버네이트 프로젝트 설정은 쉬운 편이다. 여기서는 넷빈즈NetBeans IDE에서 개발된 메이븐Maven 기반의 프로젝트 소스를 사용한다. 개발환경 설정에 관해 자세히 설명하지는 않지만, 다음 단계를 따라 진행하면 도움이 될 것이다. 비록 넷빈즈 IDE에서 개발된 코드지만, 기호에 맞는 다른 IDE를 사용해도 괜찮다. 또한, 더비나 HyperSQL과 같은 인-메모리In-Memory 데이터베이스로 바꿔 사용해도 괜찮다.

우선 JDK 5.0+, 넷빈즈 IDE, 메이븐, MySQL 데이터베이스로 구성된 필수 개발환경을 준비한다. 여기서는 JDK 6, 넷빈즈 7.3, 메이븐 3.2, MySQL 5.2, Hibernate 4.2.3.Final을 사용했다.

개발환경이 준비되었다면, 다음 단계는 넷빈즈(또는 이클립스) IDE에서 새 메이븐 프로젝트를 생성하는 것이다. 다음과 같이 pom.xml 파일에 적당한 하이버네이트 라이브러리 의존성dependencies을 추가하자.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
          http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.madhusudhan</groupId>
<artifactId>just-hibernate</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>4.2.3.Final</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.18</version>
    </dependency>
    ...
</project>
```

메이븐은 프로젝트를 빌드할 때 관련 라이브러리 의존성을 해결해 준다. 제공하는 소스코드를 내려받고, 각자 주로 사용하는 IDE로 프로젝트를 가져오길 바란다.

다음 단계로 데이터베이스를 준비한다. 데이터베이스가 이미 있다면 이 단계는 넘어간다. 여기서는 MySQL 데이터베이스를 사용하므로 최신 MySQL 패키지를 설치한다. MySQL(또는 다른 데이터베이스)이 설치되었다면 다음과 같이 'JH'를 스키마 명으로 생성한다.

```
CREATE SCHEMA JH;
```

프로젝트에서 테이블 대부분은 하이버네이트에 의해 자동으로 생성된다. 하이버네이트는 상황에 따라(테이블이 없다면) 설정 파일을 읽고 테이블을 생성한다. 테이블 자동생성을 위해 하이버네이트 설정 파일(hibernate.cfg.xml)의 속성(hbm2ddl.auto)에 다음과 같이 값을 정의한다.

```
<property name="hbm2ddl.auto">update</property>
```

이 속성값을 설정할 때 테이블이 없으면 자동으로 테이블이 생성되고, 테이블 스키마의 변경이 생기면 자동으로 반영된다.

`hbm2ddl.auto` 속성을 상용화된 서비스에 사용해서는 안 되며 모든 테이블 정의를 통해 스키마를 생성해야 한다.

여기까지 하이버네이트의 기초에 대해 알아보았다.

다루기 힘든 JDBC 구문과 커넥션의 핵심 부분을 숨기는 메커니즘을 기다려왔고 번거로운 컬럼 설정setting/getting 없이 POJO 객체를 데이터베이스로 바로 저장할 수 있는 기능을 만들고 싶었다. 하이버네이트를 도입함으로써 그동안 바라던 것을 이루게 되었다. 궁금한 것이 많겠지만 하이버네이트와 함께 하는 여정을 통해서 이런 궁금증이 해결될 것이니 계속 함께 가자.

1.8 요약

이 장에서는 예제를 통해 객체-관계-모델object-relational-model 문제를 살펴보았다. 데이터 접근을 위해 JDBC를 사용할 수 있지만, 많은 매핑 수작업과 불필요하게 반복되는 코드가 요구된다는 것을 알게 되었다. 객체-관계object-to-relational의 데이터 영속화 문제를 풀기 위해 몇 단계를 거쳤고, 하이버네이트를 소개했다. `SessionFactory`와 `Session`에 관한 하이버네이트의 고급 개념도 살펴보았다. 하이버네이트 프레임워크로 JDBC 예제를 리팩토링했다. 그리고 예상과 같이 POJO를 영속화하고 조회하는 데 성공했다.

다음 장에서는 하이버네이트의 기본 개념에 대해 좀 더 자세히 살펴보겠다.

2 | 기본 개념

앞 장에서 하이버네이트 프레임워크를 이용하여 Movie 객체를 저장하는 기본 예제를 살펴보았다. 세세한 구성 요소에 신경 쓰지 않고 하이버네이트를 짧은 시간에 살펴보는 것이 목표이므로 세부 내용은 건너뛰었다. 이제 프레임워크의 기본 개념을 살펴볼 차례다.

하이버네이트의 기본 개념을 알아보기 전에 먼저 이해하고 넘어갈 것이 있다. 그것은 객체-관계 모델의 불일치라는 매우 중요한 문제다(ORM이 먼저 개발된 중요한 이유이기도 하다).

2.1 객체-관계 간 모델 불일치

객체-관계 영속성object-relational persistence 세계에서 객체-관계object-relational 패러다임의 불일치에 대한 이해 없이는 어떠한 논의도 할 수 없다. 이 패러다임을 이해함으로써 ORM 툴의 탄생 배경과 현재까지 사용되는 이유에 대한 통찰을 얻을 수 있다.

객체 세계는 항공편 예약, 현금 이체, 도서 구매 등 특정 상황의 문제를 해결하기 위해 만들어졌다. 이를 해결하기 위한 기술적인 해결책은 도메인 모델을 중심으로 설계되었다. 도메인 모델은 현실 세계의 문제를 대변하는 객체들을 의미한다. 항공 예약, 은행 계좌, 도서 조회 등이 도메인 객체의 예라고 할 수 있다. 이런 도메인 객체는 데이터베이스에 저장되고 찾아서 읽어 올 수 있다.

현실 세계의 애플리케이션 대부분은 영속성 형태 없이는 존재할 수가 없다. 거래 은행으로부터 예금이 저장 공간에 지속해서(장기간) 보관되지 않아 유실되었다는 소식을 듣는다면 큰 혼란에 빠질 것이다. 또한, 구매한 항공권이 출국 직전에 예약이 안 되었다고 확인된다면 항공사에 매우 큰 불만을 가질 것이다. 그러므로 관계형

데이터베이스의 영속성은 지속적인 스토리지의 산업 표준이 되었다.

관계형 데이터베이스에는 로우와 컬럼의 2차원 형태로 데이터가 저장된다. 데이터 관계는 외래키foreign key 형태로 표현된다. 문제는 도메인 객체를 관계형 데이터베이스로 저장할 때 발생한다. 애플리케이션의 객체는 로우와 컬럼 형태가 아니다. 도메인 객체는 객체의 상태를 속성(변수)으로 가지고 있다. 그래서 도메인 객체 그대로 관계형 데이터베이스에 저장할 수가 없다. 이러한 불일치를 객체-관계 간 임피던스 불일치object-relational impedance mismatch라고 한다.

객체와 관계형 모델 사이에는 몇 가지 근본적인 차이가 있는데, 간단히 살펴보자.

2.1.1 상속 불일치

상속은 객체 세계에서는 지원하지만, 관계형 스키마에서는 지원하지 않는다. 상속은 모든 객체지향 언어, 특히 자바에서 바늘과 실처럼 뗄 수 없는 특징이다. 안타깝게도 관계형 스키마에는 상속 개념이 없다. 회사에서 임원과 직원의 예를 들어보자. 임원 개인도 회사의 직원이다. 이 관계를 데이터베이스에서 표현하는 것은 테이블 간 관계 수정이 필요해서 쉽지 않다.

상속 없이 현실 세계의 문제 상황을 표현하는 것은 매우 복잡한 일이다. 그런데 데이터베이스는 상속 관계와 같은 형태를 알지 못한다. 이것을 해결할 간단한 방법은 없지만, 문제를 풀 수 있는 몇 가지 접근법이 있다. 이 접근법은 다음 장에서 살펴볼 다양한 클래스-테이블class-to-table 전략을 사용한다. 그리고 데이터베이스에서 사용하는 has a 관계도 살펴보겠다.

2.1.2 동일성 불일치

자바 애플리케이션에서 객체는 동일성Identity과 동등성Equality의 특징을 모두 가지고 있다. Trade POJO의 선언부를 살펴보자.

```
public class Trade {
     private long tradeId = -1;
     private double quantity = 0;
     private String security = null;
…
```

두 객체는 동일한 메모리 영역(주소)을 가리키므로 동일한 객체로 여겨진다.

```
Trade trade1 = new Trade();
Trade trade2 = trade1;

// Memory location is identical—identity check!
if(trade1==trade2){ … }

// Values are identical—equality check!
if(trade1.equals(trade2)){ … }
```

'==' 연산자로 두 Trade 객체를 비교할 때 메모리 영역을 기준으로 비교하는데, 이 연산자는 두 Trade 객체의 메모리 주소값을 비교한다. 두 객체가 같은 주소값을 가지고 있다면(trade1, trade2처럼) 두 객체는 동일identical하다.

하지만 한 객체(trade1)의 값이 다른 객체(trade2)의 값과 같다면 두 객체는 동등equals하다. 앞 코드의 두 번째 코드 블록과 equals 메소드를 이용한 코드 블록은 동등성을 보여준다. 개발자는 동등성을 정확하게 표현할 수 있도록 equals 메소드를 작성할 의무가 있다.

자바에서는 '==' 연산자와 equals 메소드에 차이가 있다. 동일한 객체인지 확인하기 위해 '==' 연산자를 사용하고, 같은 값의 객체인지 확인하기 위해 equals 메소드를 사용한다.

그러나 관계형 스키마에서는 동일성과 동등성의 개념이 없다. 각 로우(또는 레코드)는 컬럼의 값을 통해 구분한다. 데이터베이스는 동일성과 동등성을 구현하는 데 부족함을 채우기 위해서 기본키Primary Key 전략을 사용하기도 한다. 기본키 식별자는 구분된 각각의 로우를 나타내고, 동일성과 동등성의 개념을 나타내기 위한 객체 속성의 하나로 활용된다. 예를 들어, trade의 `tradeId` 속성은 `TRADES` 테이블의 `TRADE_ID` 기본키에 해당한다.

2.1.3 관계와 연관 관계의 불일치

자바와 같은 객체지향 언어에서 연관 관계Associations는 중요한 특징이다. 현실 세계에서 장바구니는 구매 품목의 집합이고, 자동차는 차량등록, 제조, 모형, 그리고 엔진 등의 속성을 가지고 있다. `Trade`는 상품, 수량, 만기일 등의 속성이 있다. 이런 관계는 자바에서 연관 관계 형태로 모두 표현된다. 보통 자바 애플리케이션에서는 다양한 연관 관계를 찾아볼 수 있다.

다행히 관계형 데이터베이스에서도 연관 관계를 어느 정도 구현할 수 있다. 다른 테이블의 기본키를 참조하는 외래키는 관계형 스키마에서 연관 관계를 표현하는 방법이다. 하지만 자바에서 객체 모델은 한 타입이 아닌 세 가지 타입(일대일, 일대다 또는 다대일, 다대다)의 연관 관계를 보여준다. 여러 타입의 연관 관계를 외래키/기본키 관계를 변환하는 작업은 불가능하진 않지만 무척 어려운 일이다.

ORM 툴은 우연이 아닌 필요에 의해 탄생되었다. 객체와 관계형 모델 사이를 연결하는 일은 많은 시간과 노력이 필요하다. 하이버네이트와 같은 ORM 툴은 객체-관계 영속성 전략object-relational persistence strategies 과정을 자동화하기 위해 개발되었다.

2.2 하이버네이트의 핵심

하이버네이트 애플리케이션의 가장 중요한 특징은 다음과 같다.

- 영속성 클래스(POJO, AJO[Annotated Java Objects])
- 설정과 매핑 정의
- API를 통한 데이터 접근과 제어

도메인 모델은 객체 영속성을 말한다. 도메인 객체는 애플리케이션 내에서 영속화된 클래스의 역할을 한다. 설정 정보는 어떤 객체가 저장되고 어느 위치에 저장되는지 알려준다. 결국, 하이버네이트 API로 데이터를 가져오고 저장하고 삭제하고 변경한다

다음 절에서는 데이터 제어에 대해 자세히 살펴보겠다.

2.3 영속화된 클래스들

비즈니스 애플리케이션의 기본 요건은 애플리케이션에서 생성된 데이터의 지속성이다. 예를 들어, 한 편의 영화를 나타내는 `Movie` 객체를 생성한 JustMovies 애플리케이션이 있다고 하자. JustMovies 애플리케이션이 없어지더라도 `Movie` 객체에는 접근이 가능하다.

영속적인 객체의 요건은 도메인 객체(POJO, AJO)를 생성하고, 하이버네이트로 지속 가능한 저장 공간에 객체를 저장함으로써 충족할 수 있다. 다음으로 영속성을 위한 객체 생성을 살펴보자.

2.4 예제: 거래 시스템

데이터베이스로 `Trade`(거래 정보)를 저장하고 조회하는 자바 애플리케이션을 떠올려보자. `Trade` POJO로 정의된(앞의 예제에서 `Trade` 클래스 선언 부분 참고) 각각의

Trade(거래 정보)는 TRADES 테이블의 새로운 로우로 데이터베이스에 저장된다(id는 애플리케이션을 통해 유일한 값으로 생성된다).

하이버네이트 관점에서 영속성 클래스는 특별할 것이 없다. persistence 클래스라는 이름에서 유추할 수 있듯이 영속성 클래스는 하이버네이트가 어떻게 처리할지 알고 있는 POJO(또는 AJO)다.

객체의 식별에 대해서만 논의해 보자. 하이버네이트 내에서는 모든 영속성 객체가 유일한 식별 키를 가지도록 요구한다. Trade 클래스의 tradeId 변수는 유일한 Trade 객체를 만들어 낸다. tradeId는 테이블의 기본키며, TRADES 테이블의 TRADE_ID에 저장된다.

매핑 정의를 통해서 식별 정보를 하이버네이트에 전달해야 한다. 애플리케이션에 몇 줄의 코드를 추가함으로써 식별 정보를 설정할 수 있다. 또는 데이터베이스로부터 추출하는 방법(예, 일련번호)도 있다. 식별자가 무엇이고 어떻게 생성하는지 살펴보자.

2.5 어노테이션 사용하기

일반적으로 영속성 클래스와 데이터베이스 테이블 간의 매핑은 코드 밖의 매핑 테이블을 통해 처리된다는 점에 주목하자(보통 xml 파일).

또 다른 매핑 방법도 있다. Java 5에서 소개된 어노테이션annotations은 하이버네이트를 포함한 많은 프레임워크에 빠르게 도입되었다. 클래스의 메타 정보를 포함한 어노테이션은 클래스에 소스 코드 레벨로 추가되었고, 실제 코드가 실행되는 부분에는 영향을 주지 않는다.

다음은 클래스와 변수 레벨에서 어노테이션으로 선언된 매핑 정보가 있는 Trade 클래스다.

```
@Entity
@Table(name="TRADES")
public class Trade implements Serializable {
     @Id
      private long tradeId = -1;
      private double quantity = 0;
      private String security = null;
      // Getters와 setters
         ...
}
```

각 영속성 객체는 `@Entity` 어노테이션(클래스 레벨)으로 표기한다. `@Table` 어노테이션은 데이터베이스 테이블을 선언하고, 해당 테이블에 영속성 객체들이 저장된다. 클래스명과 테이블명이 같다면 `@Table` 어노테이션을 표기하지 않아도 된다(예제에서 클래스명은 `Trade`, 테이블명은 `TRADES`). `@Id` 어노테이션은 객체 인스턴스의 유일한 식별자 변수(기본키)를 가리킨다.

POJO를 어노테이션에 맞게 변경한다면 일반적으로 지칭하는 '자바 객체'가 아닐까? 양분되어 논쟁 중이지만, 이 책에서는 '어노테이티드 자바 객체annotated Java object'라고 부르겠다.

하이버네이트는 자바 퍼시스턴스 API(JPA, Java Persistence API) 어노테이션을 사용한다. JPA는 자바 객체의 영속성을 기술하는 표준 명세다. 앞에서 다룬 어노테이션들은 `javax.persistence` 패키지에 포함되어 있다. 이 책에서는 어노테이션과 xml 설정 모두를 다룰 예정이다.

이번 절에서는 영속성 클래스를 만들어 보았다. 다음 절에서는 설정과 매핑에 대해서 살펴본다.

2.6 설정하기

데이터베이스에 연결하려면 하이버네이트 프레임워크는 데이터베이스의 URL, credentials, dialects 등의 정보가 필요하다. 애플리케이션이 구동되면 하이버네이트 프레임워크는 설정값을 통해 데이터베이스와 연결을 시작한다.

하이버네이트 데이터베이스 설정은 보통 일회성이다. 1장에서 살펴본 것과 같이 하이버네이트 설정은 속성 파일(hibernate.properties)이나 XML 파일(hibernate.cfg.xml)로 제공된다. 사용자에 따라 다른 파일명으로 사용할 수 있지만, 사용될 파일을 명확하게 선언해야 한다. 그리고 명시된 파일명이 없이 하이버네이트를 시작하면 클래스패스를 통해 기본 설정 파일들을 모두 로딩한다.

2.6.1 속성 파일 사용하기

데이터베이스 연결 속성은 hibernate.properties 파일의 '이름/값' 쌍으로 제공된다.

```
#MySQL Properties
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://localhost:3306/JH
hibernate.connection.username = myuser
hibernate.connection.password = mypassword
hibernate.dialect = org. hibernate.dialect.MySQL5Dialect
```

이 속성들은 MySQL 데이터베이스를 가리킨다. 다른 데이터베이스를 사용한다면 driver_class, url 등의 속성값들을 알맞게 변경해야 한다. 예를 들어, 다음은 더비를 사용했을 때 연결 정보를 보여준다.

```
#Derby Properties
hibernate.connection.driver_class = org.apache.derby.jdbc.EmbeddedDriver
```

```
hibernate.connection.url = jdbc:derby:memory:JH;create=true
hibernate.connection.username = myuser
hibernate.connection.password = mypassword
hibernate.dialect = org.hibernate.dialect.DerbyDialect
```

애플리케이션의 클래스패스에서 벤더에서 제공하는 드라이버 클래스들을 사용하는지 반드시 확인해야 한다. 메이븐 기반 프로젝트라면 pom.xml 파일에 라이브러리 의존성을 추가할 수 있다.

이 책은 메이븐 기반 프로젝트로 진행된다. 메이븐 프로젝트를 설정하는 자세한 방법은 "1.7 하이버네이트 설정하기"를 참고하기 바란다.

2.6.2 XML 파일 사용하기

속성 파일을 사용하는 대신에 메타데이터를 XML 파일에 선언해서 사용할 수 있다.

```
<?xml version="1.0" encoding="utf-8"?>
<hibernate-configuration>
     <session-factory>
        <property name="connection.url">jdbc:derby:memory:JH;create=true</
property>
        <property name="connection.driver_class">
     org.apache.derby.jdbc.EmbeddedDriver</property>
   <property name="connection.username">jhuser</property>
   <property name="dialect">org.hibernate.dialect.DerbyDialect</property>
  </session-factory>
</hibernate-configuration>
```

SessionFactory 객체는 설정 파일의 속성을 이용하여 생성하므로 session-factory 태그 하위에 선언된다. 또한, hibernate.properties 파일에 정의된 속성들과 같이 hibernate.*로 시작되는 속성도 이곳에 추가한다.

2.6.3 설정 속성

하이버네이트를 실행할 때 속성이 어떻게 적용되는지 알아볼 차례다. [표 2-1]은 설정에 필요한 중요한 속성들이다.

[표 2-1] 설정 속성

속성	값	설명
hibernate.show_sql	true/false	참이면 모든 SQL문 출력
hibernate.jdbc.fetch_size	〉=0	JDBC 조회 크기 설정
hibernate.jdbc.batch_size	〉=	쿼리문 일괄 실행 시 사용
hibernate.hbm2ddl.auto	validate/update/create/create-drop	스키마 자동생성 옵션
hibernate.connection.pool_size	〉=1	커넥션 풀 크기

설정 방법(속성 파일과 XML 파일)에 관한 선택만 남았다. 어떤 방법을 사용해야 할지 궁금할 것이다. 꼭 지켜야 하는 명확한 규칙이 있는 것은 아니고, 두 가지 방법 중에서 원하는 것을 사용하면 된다. 사실 두 종류의 설정 파일이 모두 존재하더라도 하이버네이트는 정상 작동하지만, 속성 파일은 그냥 무시된다. 즉, XML 파일의 속성들은 속성 파일보다 우선한다.

2.6.4 프로그래밍 설정

앞의 두 가지 설정 방법(속성 파일, XML 파일)은 선언하는 방식이며, 또한 프로그래밍 설정도 지원한다. 이때 다음과 같이 Configuration 클래스 인스턴스를 생성하는 클래스를 이용한다.

```
Configuration cfg = new Configuration()
      .setProperty("hibernate.dialect", "org. hibernate.dialect.DerbyDialect")
      .setProperty("hibernate.connection.username", user);
      .setProperty("hibernate.connection.password", password);
      .setProperty("hibernate.connection.url", "jdbc:derby:memory:JH;create=true")
      .setProperty("hibernate.order_updates", "true");
```

Configuration 클래스 대신 매개변수로 표준 가상머신Standard VM 인수 형식을 사용할 수도 있다.

```
-Dhibernate.connection.url=jdbc:derby:memory:JH;create=true
-Dhibernate.username=mk
```

2.7 매핑

다음은 어떤 종류의 객체가 영속화되는지를 살펴봐야 한다. 그리고 어떻게 객체의 필드가 테이블의 컬럼과 매핑되고, 각 필드의 영속화를 어디에서 제어하는지도 살펴봐야 한다.

매핑은 객체-관계형 데이터를 만들기 위해 하이버네이트 프레임워크가 이해할 수 있는 메타데이터다. 앞에서처럼 XML 파일이나 어노테이션을 이용하여 메타데이터 매핑을 선언할 수도 있다.

2.8 XML 매핑 파일

.hbm.xml 확장자를 가진 XML 파일에 명시된 메타데이터를 이용하여서 도메인 POJO를 선언한다. 이 파일은 하이버네이트에서 매핑 정의를 로딩할 수 있도록 클래스패스에 생성한다. 예를 들어, Trade 객체는 Trade.hbm.xml 파일에, Movie 객체는 Movie.hbm.xml에 선언되어야 한다.

꼭 한 개의 객체를 한 개의 매핑 파일에 정의할 필요는 없다. 사실, 매핑 파일 한 개에 모든 모델을 정의해도 된다. 그렇긴 해도 각 영속화 객체에 맞는 개별 매핑 파일을 사용하길 바란다. 이는 매핑 정의와 코드를 좀 더 쉽게 유지보수할 수 있기 때문이다. 특히 도메인 모델이 점점 복잡해질수록 효과적이다. Trade.hbm.xml 매핑 정의 파일은 다음과 같다.

```
<hibernate-mapping>
    <class name="com.madhusudhan.jh.domain.Trade" table="TRADES">
        <id name="tradeId" column="TRADE_ID">
            <generator class="assigned"/>
        </id>
        <property name="quantity" column="QUANTITY"/>
        <property name="security" column="SECURITY"/>
    </class>
</hibernate-mapping>
```

매핑 정의 파일에서 몇 가지 더 살펴보자.

클래스-테이블 매핑은 앞 코드의 class 태그를 통해 이루어진다. class 태그 선언부에는 Trade 객체가 TRADES 테이블로 영속화된다고 명시되어 있다. 테이블 속성 정의 역시 객체가 어디에 저장될지 보여준다. 그리고 클래스-테이블 매핑 정의를 선언한 후 영속화될 객체 속성을 하이버네이트에 알려준다.

영속화 속성 리스트 중에서 첫 번째이자 가장 중요한 속성은 객체의 식별자(기본키)다. tradeId는 id 태그를 이용하여 테이블 기본키인 TRADE_ID와 연결된다. id 태그의 name 속성은 Trade 클래스의 tradeId 변수와 일치한다. 하이버네이트에서는 TRADE_ID 변수에 맞게 값을 설정하거나 가져오기 위해 getTradeId와 setTradeId 메소드를 호출한다.

Trade 객체에서 영속화될 필드는 property 태그로 정의된다. 앞 예제에서 Trade 객체의 quantity 필드는 column 태그로 정의된 QUANTITY로 저장된다. 그리고 security 값은 SECURITY 컬럼에 매핑된다.

좀 더 쉽게 매핑하는 방법이 있다. 객체의 변수명과 테이블 컬럼명이 일치한다면 컬럼 속성 선언은 생략해도 된다. 앞의 예제에서 변수명이 컬럼명과 일치하므로 quantity와 security의 column 속성은 빼도 된다.

```
<hibernate-mapping>
    <class name="com.madhusudhan.jh.domain.Trade" table="TRADES">
        ...
        <property name="quantity"/>
        <property name="security"/>
    </class>
</hibernate-mapping>
```

앞부분에 속성의 데이터 타입을 명시하는 것을 빠뜨렸다. quantity가 double 형이고, security가 String 형인지 어떻게 알 수 있을까? 다음처럼 데이터 타입 역시 type 속성을 이용하여 명시하면 된다.

```
<hibernate-mapping>
    <class name="com.madhusudhan.jh.domain.Trade" table="TRADES">
        ...
        <property name="quantity" type="double"/>
        <property name="security" type="string"/>
    </class>
</hibernate-mapping>
```

데이터 타입을 생략하고 자바 리플렉션으로 변수 타입을 알아내거나 직접 선언해도 된다. 개인적으로는 직접 데이터 타입을 선언하는 것을 선호한다. 이는 애플리케이션이 시작될 때 데이터 타입을 찾기 위해 낭비되는 시간을 줄일 수 있다.

앞의 아주 간단한 매핑 예제와 달리 연관 관계, 관계, 쿼리, 다른 하이버네이트 요소 등을 추가하여 좀 더 많은 것들을 할 수 있다. 다음 장에서 이 주제를 다시 살펴보겠다.

설정 파일은 hibernate.cfg.xml(또는 hibernate.properties)와 Trade.hbm.xml(도메인 매핑 파일) 두 개다. 데이터베이스 설정에 맞게 애플리케이션이 시작할 때 매핑 관계를 알 수 있도록 설정 파일에 매핑 파일을 명시한다.

다음은 매핑 파일을 어떻게 명시하는지 보여주는 설정 파일의 일부다.

```
<hibernate-configuration>
    <session-factory>
        <property name="connection.url">
            jdbc:derby:memory:JH;create=true
        </property>
        ...
        <mapping resource="Movie.hbm.xml" />
        <mapping resource="Trade.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

2.9 식별자 생성 방법

각 객체는 유일한 식별자를 가지고 데이터베이스에 영속화되어야 한다. 이때 식별자를 자동으로 생성하는 다양한 방법을 활용할 수 있다. 하이버네이트에서는 식별자 생성과 관련하여 다양한 방법을 제공하는데, 이번 절에서 살펴보겠다. id 요소를 설정하려면 generator 클래스를 정의해야 한다.

```
<hibernate-mapping>
    <class name="com.madhusudhan.jh.domain.Movie" table="MOVIES">
        <id name="id" column="MOVIE_ID">
            <generator class="assigned"/>
        </id>
        ...
    </class>
</hibernate-mapping>
```

generator는 식별자 생성 방법을 선택하는 데 중요한 요소로, generator 요소 내 클래스 속성class attribute은 적용할 식별자 생성 방법을 가리킨다. 앞의 코드에서는 영속화된 객체마다 유일한 식별자를 정하도록 'assigned'을 사용한다.

'assigned'는 org.hibernate.id.Assigned 클래스의 줄임말이다. generator 클래스를 전체 패키지명과 함께 명시하는 대신 일반적으로 줄임말 버전을 사용한다. 예를 들어, org.hibernate.id.SequenceGenerator에 해당하는 'sequence', org.hibernate.id.IdentityGenerator에 해당하는 'identity' 등이 있다. 모든 generator 클래스는 공통 인터페이스인 org.hibernate.id.IdentifierGenerator를 구현한다.

하이버네이트에서 제공하는 generator 클래스는 identity, sequence, native, assigned가 있다. 자신만의 기본키 생성 방법이 필요하다면 IdentifierGenerator 인터페이스를 구현하고 요구에 맞게 로직을 만들어서 사용자 정의 방법을 생성할 수 있다.

param 요소로 generator 요소에 필요한 인수를 명시할 수 있다. 예를 들어, sequence 방법을 사용할 때는 하이버네이트에 sequencer 테이블에 대한 정보를 명시해야 한다.

```
<id name="id" column="MOVIE_ID">
     <generator class="sequence">
        <param name="sequence">MOVIE_SEQUENCE</param>
     </generator>
</id>
```

2.10 세션 API

하이버네이트는 객체의 영속성과 조회에 관한 풍부한 API를 가지고 있다. 모든 API를 살펴보는 것은 이 책의 범위를 벗어나지만, 하이버네이트 개념을 이해하는 데 꼭 필요한 Session API를 다루기로 한다.

이번 장에서는 하이버네이트에서 가장 기본이 되는 API 사용법을 살펴보았다. org.hibernate.SessionFactory 클래스에서 제공하는 SessionFactory는 Session 인스턴스를 생성하는 팩토리 클래스factory class다. 이는 Thread-safe한 객체이므로 데이터가 의도하지 않게 바뀌는 것을 염려하지 않고, 여러 클래스에서 사용해도 된다. Session 인스턴스가 생성될 때 매핑 정보도 함께 전달하므로 컴파일된 형태로 모든 매핑 데이터를 가진다.

이전에 본 것처럼 설정 파일명을 명시함으로써 Session 팩토리를 생성한다.

```
private SessionFactory getSessionFactory() {
     Configuration config = new Configuration()
         .configure("hibernate.cfg.xml");
     ServiceRegistry registry = new ServiceRegistryBuilder()
         .applySettings(config.getProperties()).buildServiceRegistry();
     return config.buildSessionFactory(registry);
}
```

여기서는 설정 정보를 생성하기 위해 하이버네이트(4.x 버전)에 새로 도입된 서비스 레지스트리 클래스service registry class를 이용한다.

SessionFactory는 캐시의 두 번째 수준을 관리한다. 이는 애플리케이션을 구성하는 모든 컴포넌트에도 해당된다. 전역global 캐시는 데이터베이스에서 이미 가져온 동일한 결과를 여러 애플리케이션에서 요청하는 경우 사용된다. 이는 애플리케이

션 요청 시간을 단축시킨다.

SessionFactory가 데이터베이스 연결을 위한 열쇠 꾸러미를 가지고 있다면, Session은 데이터베이스 접속과 데이터 이동이 이루어지는 열쇠 자체다. Session은 싱글 스레드single-threaded 객체이므로 여러 컴퍼넌트에 같이 선언되어 사용해서는 안 된다. 하나의 작업 단위를 뜻한다. 팩토리에서 세션을 가져오려면 factory.getCurrentSession() 메소드를 사용한다. 세션 객체를 얻으면 한 개의 트랜잭션 안에서 데이터베이스 작업을 수행한다. 세션과 트랜잭션은 밀접한 관련이 있다. 다음은 세션과 트랜잭션의 생애 주기를 보여주고 있다.

```
// 현재 세션 가져오기
Session session = factory.getCurrentSession();
// 트랜잭션 생성하고 시작하기
Transaction tx = session.getTransaction();
tx.begin();
// 세션을 가지고 데이터베이스 조작하기
// 조작이 완료되었다면, 트랜잭션과 세션 커밋하기
tx.commit();
```

캐시의 첫 번째 수준는 세션 내에서 유지된다는 것을 보여준다(조회해서 가져오거나 접근하는 모든 객체는 세션이 닫히기 전까지 세션 내에서 유지된다). 데이터베이스를 작업할 때마다 세션을 사용하지는 않는다. 하나의 트랜잭션 내에서 연관된 모든 데이터베이스 작업이 이루어지는 것이 좋다.

2.11 트랜잭션

트랜잭션은 각 작업이 분리된 상태로 수행되도록 하고, 수행되는 동안 잘못된 데이터가 읽히거나 쓰여지지 않도록 동기화한다. 데이터베이스 트랜잭션에서 기준으로 삼는 네 가지 기본 속성이 있다. Atomicity(원자성), Consistency(일관성),

Isolation(격리성), Durability(지속성)이며, ACID 속성으로 부르기도 한다. 트랜잭션을 이해하면 소프트웨어 설계에 큰 도움이 된다.

런던 파리 간 열차표를 예매하는 예제를 살펴보자. 결제가 진행될 때 결제 승인과 좌석 배정이 한번에 모두 이루어지길 원한다. 예매 과정에서 문제가 발생해서 예매가 되지 않았는데도 결제가 되는 것을 원하지는 않는다. 그러므로 앞선 과정이 성공적으로 이루어져야 순차적으로 다음 과정으로 넘어가며, 어느 과정에서 실패하면 모든 과정이 취소되어야 한다. 단계적으로 이어지는 모든 단계는 트랜잭션 범위에 포함된다.

일반적으로, 조회해서 가져오는 트랜잭션에는 두 가지 방법이 있다. 컨테이너container에서 트랜잭션을 생성하고 관리하는 방법과 사용자가 작성해서 트랜잭션을 관리하는 방법이다. 전자의 경우 컨테이너에서 모두 관리하기 때문에 커밋과 롤백 같은 트랜잭션에 대해 고민할 필요가 없다.

standalone JVM에서는 독립적으로 트랜잭션을 사용한다. 다음의 `Course` 객체를 저장 공간에 영속시키는 부분을 살펴보자.

```
private void persist() {
     Transaction tx = null;
     try {
        // 트랜잭션 인스턴스 생성하기
        tx = session.beginTransaction();
        // Course 인스턴스 생성과 영속 작업 시작하기
        Course course = createCourse();
        session.save(course);
        // 미션 완료
        tx.commit();
     } catch (HibernateException he) {
        // 문제 발생, 미션 취소
```

```
            if(tx!=null)
            tx.rollback();
            throw he;
        } finally {
            // 트랜잭션을 반드시 닫아야 한다
            session.close();
        }
    }
```

새로운 Transaction 객체를 생성하고 반환하는 session.beginTransaction 메소드를 호출해서 트랜잭션을 시작한다. 세션과 연계된 트랜잭션 객체는 커밋 또는 롤백될 때까지 활성화된다.

트랜잭션 내에서 작업을 수행하고 나면 동일 트랜잭션에서 커밋한다. 이 단계에서 엔티티는 데이터베이스로 영속화된다. 영속화되는 동안 에러가 발생하면 실행 중인 하이버네이트에서는 발생한 에러를 예외처리하고, HibernateException(unchecked RuntimeException)를 발생시킨다. 이렇게 되면 예외를 처리하고 트랜잭션을 롤백해야 한다. 사용자에게 정보 제공 또는 디버깅의 목적을 위해서 추가적인 예외를 발생시키기도 한다.

2.12 요약

이번 장에서는 하이버네이트를 구성하는 핵심 요소를 다루었다. 객체-관계형 패러다임 불일치와 이를 해결할 하이버네이트와 같은 ORM 툴을 살펴보았다. 또한, 하이버네이트 설정 클래스와 데이터베이스-테이블 매핑, 영속성 클래스 설계에 관해 상세히 살펴보았다.

3 | 어노테이션

어노테이션은 Java 5에서 도입된 이후로 대부분의 툴킷toolkits에서 많이 활용되기 시작했으며, 클래스와 변수 레벨에서 메타 정보를 정의하기 위해 사용하는 데코레이터decorator다. 하이버네이트는 어노테이션을 수용하여 최상위 객체로 통합하였다. 어노테이션은 세련되게 ORM 매핑 개발을 할 수 있는 필수 도구이므로 이 장에서 자세히 살펴보겠다. 또한, 어노테이션의 사용 방식과 전통적인 XML 파일 방식의 장단점도 알아본다.

이전 장에서는 XML 파일을 이용한 영속성 클래스 매핑을 살펴보았는데, 설정과 매핑을 위한 XML 파일에는 장단점이 있다. 우선, XML 파일은 아주 간단하고 가독성이 좋으나 너무 장황하며 타입 안전성을 보장받기 어렵다. 하지만 어노테이션은 매우 간결하고 컴파일 시 타입 체크를 바로 할 수 있다. 이는 클래스에 직접 사용할 수 있는 메타 데코레이션이므로 엔티티를 효과적으로 관리할 수 있다.

이제 어노테이션에 관해 자세히 살펴보자.

3.1 예제를 통해 실행하기

간단한 예제로 어노테이션을 배우면 이해가 빠를 것이다. 자, 어노테이션 없이(순수한 POJO) 정의된 간단한 영속성 클래스 Employee와 함께 시작해 보자.

```
public class Employee {
     private int id =0;
     private String name = null;
     …
}
```

이 클래스를 영속화하려면 먼저 엔티티로 정의해야 하는데, @Entity 어노테이션으로 정의할 수 있다. 간단하지 않은가?

```
@Entity
public class Employee {
     private int id =0;
     private String name = null;
     …
}
```

영속성 엔티티를 만들었으니 식별자를 정의해 보자. 모든 영속성 엔티티는 식별자를 정의해야 한다는 것을 기억하자. 그렇지 않으면 하이버네이트는 결국 예외를 발생시킬 것이다. 그래서 객체의 식별자를 프레임워크에 알려주기 위해 @id 어노테이션을 사용한다. Employee 클래스에서 기본키는 id 변수이므로 id 변수에 관련 어노테이션을 선언한다.

```
@Entity
public class Employee {
     @Id
     private int id = 0;
     private String name = null;
     public int getId() {
         return id;
     }
     …
}
```

id 변수에 @Id 어노테이션을 선언할 때 하이버네이트에서는 Employee 클래스의 id 변수를 EMPLOYEE 테이블의 ID 필드로 매핑한다.

getter 메소드 대신 변수에 어노테이션을 표기한 것을 보았는가? 하이버네이트가 데이터베이스 필드에 접근할 때 변수에 대한 어노테이션을 사용하거나 accessor 메소드를 사용하면 accessor 메소드에 대한 어노테이션을 사용한다.[01]

한 가지 더 살펴볼 것은 대부분의 데이터베이스에서 예약어 ID 이름으로는 필드 생성을 하지 못하므로 여기서는 EMPLOYEE 테이블의 EMPLOYEE_ID 필드명으로 변경해 준다. 객체의 변수명과 테이블의 컬럼명이 일치하지 않으면 다음과 같이 @Column 어노테이션을 추가하여 테이블 컬럼명을 정확히 명시해야 한다.

```
@Entity
public class Employee {
     @Id
     @Column(name = "EMPLOYEE_ID")
     private int id = 0;
     …
}
```

Entity와 Id 어노테이션 옵션을 알아보기 전에 우선 예제가 어떻게 동작하는지 살펴보자.

하이버네이트 설정에서 어노테이션이 선언된 클래스를 등록해야 하는데, 매핑 파일에 명시하는 방법과 클래스에 프로그래밍하는 방법이 있다.

hibernate.cfg.xml 파일에 선언된 클래스를 명시하기 위해서 mapping 요소를 이용한다. 다음은 매핑 파일을 이용한 등록 방법이다.

```
〈hibernate-configuration〉
```

01 꼭 정해진 규칙이 있는 것은 아니며, 대부분 개인의 기호에 따라 사용한다. 필자는 변수에 정의하는 것을 선호해서 이 책에서는 주로 변수에 표기한다.

```
        ...
        <mapping class = "com.madhusudhan.jh.annotations.Employee"/>
    </hibernate-configuration>
```

어노테이션 사용을 선호한다면 다음과 같이 addAnnotatedClass 메소드를 이용하면 된다.

```
Configuration config = new Configuration()
        .configure("annotations/hibernate.cfg.xml");
        .addAnnotatedClass(Employee.class)
        .addAnnotatedClass(Director.class);
...
```

Configuration 클래스에 있는 일련의 메소드를 살펴보라. 해당 클래스에서 추가 메소드를 포함시키는 편리한 방법이다.

테스트 클래스를 실행해보면 어떠한 에러도 없이 Employee 객체는 영속화된다.

3.2 자세히 살펴보기

이제 어노테이션의 주요 부분을 살펴보자.

Employee 예제에서는 테이블명과 클래스명이 동일(EMPLOYEE와 Employee)하여 @Entity 어노테이션을 선언할 때 테이블명을 명시하지 않았다. 테이블명을 TBL_EMPLOYEE로 한다면 @Table 어노테이션을 추가하여 테이블명을 명시해야 한다.

```
@Entity
@Table(name = "TBL_EMPLOYEE")
public class Employee {
        ...
}
```

id는 변수명과 컬럼명이 다르면 @Column 어노테이션을 이용하여 컬럼명을 명시해야 한다.

```
@Entity
@Table(name = "TBL_EMPLOYEE")
public class Employee {
      …
      @Column(name = "EMPLOYEE_NAME")
      private int name = 0;
}
```

또한, @Column 어노테이션 속성을 이용하여 각 컬럼의 부가 옵션 정보를 설정할 수 있다. 예를 들어, non-null 데이터 속성을 반영하는 컬럼은 nullable=false로 설정하여 해당 옵션을 적용할 수 있다. 고유성unique 제약 조건으로 생성할 컬럼은 unique=true 속성을 명시하면 된다. @Column 어노테이션 속성은 다음과 같다.

```
@Entity
@Table(name = "TBL_EMPLOYEE")
public class Employee {
      @Id
      @Column(name = "EMPLOYEE_ID", nullable = false, unique = true)
      private int empoyeeId = 0;
      …
}
```

3.3 ID 생성 방법

하이버네이트에서는 XML 매핑에서 살펴본 것처럼 어노테이션을 이용한 여러 식별자 생성 방법을 제공한다. 앞의 예제에서는 ID 생성 방법을 명시하지 않았는데,

이는 AUTO를 기본으로 사용했다는 것을 뜻한다. AUTO의 경우, 하이버네이트는 기본키 생성을 데이터베이스에 의존한다. 예를 들어, MySQL에서 AUTO_INCREMENT 옵션으로 기본키를 정의하면 하이버네이트도 해당 옵션을 그대로 사용한다.

id 변수에 @GeneratedValue 어노테이션을 추가하면 요구 사항에 맞춘 다른 생성 방법을 설정할 수 있다. @GenerateValue 어노테이션은 strategy와 generator 두 가지 속성이 있는데, strategy 속성은 사용할 식별자 생성 타입을 가리키고 generator 속성은 식별자를 생성할 메소드를 정의한다.

다음 코드는 ID 생성을 위한 IDENTITY 방법을 보여준다.

```
@Entity(name = "TBL_EMPLOYEE")
public class Employee {
     @Id
     @Column(name = "ID")
     @GeneratedValue(strategy = GenerationType.IDENTITY)
     private int employeeId =0;
     …
}
```

IDENTITY 방법을 사용하려면 generator() 메소드를 제공해야 한다. 이는 데이터베이스에서 제공하는 identity 함수에 의존하기 때문이다. MySQL, Sybase, DB2를 포함한 몇몇 데이터베이스는 identity 함수를 지원한다.

생성 방법은 다음과 같이 GeneratorType 값으로 명시해야 한다.

GeneratorType.AUTO

기본 방법으로 다른 데이터베이스 간에도 이용할 수 있다. 하이버네이트에서는 데이터베이스를 기반으로 적절한 ID를 선택한다.

GeneratorType.IDENTITY

이 설정은 몇몇 데이터베이스에서 제공하는 identity 함수를 기반으로 한다. 데이터베이스에서 고유한 식별자를 제공하는 역할을 한다.

GeneratorType.SEQUENCE

몇몇 데이터베이스에서는 연속된 숫자에 관한 메커니즘을 제공하는데, 하이버네이트에서는 일련번호를 사용한다.

GeneratorType.TABLE

다른 테이블의 고유한 컬럼 값에서 기본키를 생성하는데, 이 경우 TABLE 생성자를 사용한다.

시퀀스seqeunce 방법에서는 strategy와 generator 속성을 모두 정의해야 한다.

```
public class Employee {
      @Id
      @Column(name = "EMPLOYEE_ID")
      @GeneratedValue (strategy = GenerationType.SEQUENCE, generator = "empSeqGen")
      @SequenceGenerator(name = "empSeqGen", sequenceName = "EMP_SEQ_GEN")
      private int employeeId = 0;
      …
}
```

strategy 속성은 SEQUENCE로 정의되었고 generator 속성은 데이터베이스 시퀀스 객체를 참조하는 empSeqGen(sequence generator 참조값)으로 정의되었다. @SequenceGenerator 어노테이션을 이용하여 데이터베이스에 시퀀스 객체로 생성된 EMP_SEQ_GEN 테이블을 참조한다.

기본키를 제공하는 데이터베이스 테이블은 모든 시퀀스 속성과 함께 @TableGenerator 어노테이션을 이용하면 된다.

```
public class Employee {
      @Id
      @Column(name = "ID")
      @GeneratedValue (strategy = GenerationType.TABLE, generator = "empTableGen")
      @TableGenerator(name = "empTableGen", table = "EMP_ID_TABLE")
      private int empoyeeId = 0;
      …
}
```

3.4 복합 식별자

고유한 로우를 식별하는 기본키로 항상 단일 컬럼(대리키surrogate key)만을 갖지는 않는다. 때로 흔히 복합키composite key나 조합키compound key로 불리는 비즈니스 키를 제공하는 컬럼 조합을 갖기도 한다. 이 경우, 적절한 객체 식별자를 설정하기 위해 다른 메커니즘을 사용할 필요가 있다.

복합 아이디composite-id 식별자 설정과 관련된 세 가지 방법을 살펴보자.

3.4.1 기본키 클래스와 @Id 어노테이션 사용하기

비즈니스 키를 표현할 별도의 클래스를 생성한다. 이 클래스는 `@Embededdable` 어노테이션을 명시하고 복합 아이디 클래스를 만든다.

다음 예제의 `CoursePK` 클래스는 `tutor`, `title` 두 개의 변수로 구성되는데, 두 속성의 조합은 `Course` 클래스를 위한 복합키가 된다.

```
@Embeddable
public class CoursePK implements Serializable {
      private String tutor = null; private String title = null;
      // Default constructor
```

```
    public CoursePK() {
    }
    …
}
```

CoursePK 클래스가 @Embeddable 어노테이션과 함께 표기된 것에 주목하자.

기본 생성자default constructor와 java.io.Serializable 인터페이스를 구현해야 한다. 또한, 하이버네이트가 고유성을 구분할 수 있도록 hashCode()와 equals() 메소드가 구현되어야 한다.

다음으로 이 복합 아이디 클래스를 영속성 클래스의 id 변수에 넣어 둔다. @Id 어노테이션을 사용한 영속성 클래스는 복합 아이디 클래스를 내포하게 된다.

```
@Entity
@Table(name="COURSE_ANNOTATION")
public class Course {
    @Id
    private CoursePK id = null;
    private int totalStudents = 0;
    private int registeredStudents = 0;

    public CoursePK getId() {
        return id;
    }
    public void setId(CoursePK id) {
        this.id = id;
    }
    …
    }
```

Course 영속성 POJO 객체에는 id, totalStudents, registereredStudents라는 세 개의 변수가 있다. CoursePK 타입 id 변수는 tutor와 title 두 변수로 이루어진 복합 클래스로 Course 객체를 위한 기본 클래스가 된다.

예제를 실행해서 테이블에서 복합키가 어떻게 만들어지는지 살펴보자.

```
private void persist() {
      …
      Course course = new Course();
      CoursePK coursePk = new CoursePK();
      coursePk.setTitle("Computer Science");
      coursePk.setTutor("Prof. Harry Barry");
      course.setId(coursePk);
      course.setTotalStudents(20);
      course.setRegisteredStudents(12);
      session.save(course);
      …
}
```

결과 확인을 위해 SELECT 문을 실행해 보자.

[표 3-1] @Id 테이블 사용 결과

강좌명	강사명	등록된 학생수	전체 학생수
Computer Science	Prof. Harry Barry	12	20

3.4.2 기본키와 @EmbeddedId 사용하기

이제 복합 식별자를 생성하는 두 번째 방법을 알아보자. Course 객체의 식별자에 (앞 예제에서 추가했던 @Id 어노테이션 대신에) @EmbeddedId 어노테이션을 명시한다. 동시에 @EmbeddedId 어노테이션이 달린 내부 클래스 CoursePK를 만든다.

```
@Entity
@Table(name = "COURSE_ANNOTATION_V2")
public class Course2 {
     @EmbeddedId
     private CoursePK2 id = null;
     private int totalStudents = 0;
     private int registeredStudents = 0;
     public Course2(String title, String tutor) {
         id = new CoursePK2();
         id.setTitle(title);
         id.setTutor(tutor);
     }
     …
}
```

id 필드에 @EmbeddedId 어노테이션을 단다. 클래스 생성자 안에서 복합 기본키를 생성하고 할당하는 역할을 한다는 점을 알아두자. 물론 생성자 밖에서도 가능하다.

그러나 첫 번째 방법처럼 기본키 클래스에 @EmbeddedId 어노테이션을 명시해서는 안 된다. 기본키 클래스(CoursePK2)의 기본적인 클래스 선언(어노테이션이 없는)을 살펴보자.

```
public class CoursePK2 implements Serializable {
     private String tutor = null;
     private String title = null;
     // 기본 생성자
     // hashCode()와 equals() 메소드 구현
     @Override
     public int hashCode() { … }
     @Override
     public boolean equals(Object obj) { … }
```

기본 생성자, hashCode()와 equals() 메소드를 구현하는 기본 복합 클래스의 원칙을 지키고 있다.

객체 영속성을 위한 테스트 클라이언트를 생성하자.

```
private void persist() {
     Course2 course = new Course2("Financial Risk Management", "Harry Barry");
     course.setTotalStudents(20);
     course.setRegisteredStudents(12);
     session.save(course);
     …
}
```

title, tutor 매개변수를 가진 Course2 객체 인스턴스를 생성한다. 내부적으로는 CoursePK2 객체에서 title, tutor 변수를 설정하여 복합키를 생성한다.

테스트 클라이언트를 실행하면 테이블에 복합키와 클래스 정보가 채워지고, SELECT 문을 실행하면 첫 번째 경우와 동일한 결과가 나온다.

3.4.3 @IdClass 어노테이션 사용하기

모든 필수 기본키 속성을 가지고 컴포지트 클래스(기본키)를 만드는 방법을 알아보자. 이 클래스는 어노테이션을 하지 않으므로 기본 Java 클래스가 된다. 다음은 이 클래스의 정의 부분이다.

```
public class CoursePK3 implements Serializable {
     private String tutor = null;
     private String title = null;
     public CoursePK3() {}
     …
}
```

클래스 레벨에서 @IdClass 어노테이션을 엔티티에 선언한다.

엔티티 클래스는 @IdClass 어노테이션 방법을 따를 때 불편한 점이 생긴다. 복합 키 식별자(tutor와 title)를 클래스에 중복해서 선언해야 하고, @Id 어노테이션으로 꼭 명시해야 한다.

엔티티 클래스의 복합키 클래스 정의 부분을 살펴보자.

```
@IdClass(value = CoursePK3.class)
@Entity
@Table(name = "COURSE_ANNOTATION_V3")
public class Course3 {
    // 기본 클래스에 복합키 식별자를 중복해서 선언해야 한다
    @Id
    private String title = null;
    @Id
    private String tutor = null;
    …
}
```

복합키 선언을 위한 @IdClass 사용은 일반적인 방법이 아니며 최대한 지양하는 것이 좋다. 그 대신 앞에서 살펴본 두 가지 방법 중에서 하나를 선택하기 바란다.

POJO vs AJO

어노테이션을 이용할 때 메타데이터를 소스 클래스에 정의하므로 POJO의 본질을 일시적으로 잃게 된다. 그래서 더는 POJO가 아니라고 이야기하기도 한다. 반대로 어노테이션은 단순히 자바 마크업의 하나이므로 여전히 POJO로 분류할 수 있다고도 한다. 이 논의는 나중을 위해 남겨두고 여기서는 POJO라 부르는 대신 어노테이티드 자바 객체(AJO, Annotated Java Objects)라고 부르겠다.

3.5 요약

이번 장에서는 일반적인 어노테이션과 하이버네이트에서 어노테이션을 어떻게 지원하는지를 살펴보았다. 예제를 통해 `@Entity`, `@Id`, `@Table`, `@Column` 등 다양한 어노테이션과 알맞은 어노테이션을 활용한 식별자 생성 방법을 알아보았다.

어노테이션은 여기가 끝이 아니다. 다양한 목적을 위해 하이버네이트에서 이용되는 많은 어노테이션이 있다. XML 매핑을 이용한 어떠한 설정에서도 어노테이션을 이용할 수 있다. 이 책에서는 XML 매핑에 따른 어노테이션 사용 방법을 살펴볼 것이다.

4 | 컬렉션 영속화

공식적으로 자바 컬렉션Java collections으로 알려진 자료 구조를 가지고 작업하는 것은 자바 개발자에게는 피할 수 없는 일이다. 이번 장에서는 자료 구조를 영속화하고 조회하기 위해서 하이버네이트에서 지원하는 부분을 살펴보고, 하이버네이트 프레임워크로 어떻게 자바 표준 컬렉션를 영속화할 수 있는지 자세히 살펴본다.

한 번쯤은 자바 컬렉션을 사용한 경험이 있을 것이다. 자바 컬렉션은 다양한 알고리즘을 가진 매우 일반적인 자료 구조다. 그리고 앞으로 영속화 객체의 값으로 컬렉션을 사용하게 될 순간도 접할 것이다.

하이버네이트 프레임워크 관점에서 기본 타입 값을 영속화하는 것과 `java.util.List`, `java.util.Map` 구조와 같은 컬렉션 요소를 영속화하는 것은 다르다. 하이버네이트에 영속화를 알리려면 특정 매핑 절차와 과정을 거쳐야만 한다.

`List`, `Set`, `Array`, `Map`과 같이 이미 익숙한 자바 컬렉션들은 하이버네이트에서 모두 지원된다. 한 단계 나아가 `bag`, `idbag`과 같은 컬렉션도 생성한다. 차근차근 살펴보자.

4.1 인터페이스 설계하기

자바에서는 `java.util.Collection`(맵 인터페이스를 제외한 모든 컬렉션 인터페이스의 부모 인터페이스), `java.util.List`(리스트 자료 구조), `java.util.Set`(세트 자료 구조), `java.util.Map`(키/값 맵 자료 구조) 등의 컬렉션 인터페이스를 제공한다.

`List` 인터페이스 구현체는 요소들을 순차적으로 리스트에 담는 것이 목적이다. 보통 `ArrayList`나 `LinkedList` 같은 구현체를 사용하는데, 여기서는 `List` 인터페이

스 구현체를 사용하는 것보다 하이버네이트 애플리케이션에서 List 인터페이스를 설계하는 것이 요점이다.

컬렉션 변수 선언 시에 항상 인터페이스를 사용하라. 하이버네이트에서는 구상 클래스concrete class를 변수 타입으로 사용하는 것은 좋지 않다.

```
ArrayList<String> actors = new ArrayList<String>();
```

대신 다음과 같이 인터페이스를 이용하여서 타입을 정의해야 한다.

```
List<String> actors = new ArrayList<String>();
```

하이버네이트에서는 자체 프레임워크에서 제공하는 List 인터페이스 구현 클래스를 사용하기 때문이다.

다음으로 가장 널리 사용되는 자료 구조의 하나인 리스트에 대한 영속성 메커니즘을 살펴보겠다.

4.2 List 영속화

리스트는 순차적으로 아이템을 담는 간단하고 편리한 자료 구조다. 또한, 인덱스를 사용하여 요소의 위치를 항상 알 수 있다. 요소를 리스트의 어떤 위치에 추가해도 인덱스를 이용하여 요소를 가져올 수 있다.

예를 들어, 자동차 제조사(도요타, BMW, 메르세데스 등) 리스트를 자바 프로그램을 통해 저장했다고 생각해 보자. 입력된 순서대로 테이블에서 자동차 데이터를 조회할 수 있다. 그래서 list.get(n)을 실행한다면 별문제 없이 n번째 요소를 가져올 수 있다.

이를 위해 하이버네이트에서는 자동차 데이터의 인덱스를 가진 다른 테이블이 존재해야 한다. 이 테이블에서 자동차 정보를 조회할 때 부가적인 테이블(CAR_LIST라

고 하자)의 인덱스 순서 또한 조회된다. 해당 순서를 찾기 위해 자동차 정보 요소는 모두 연관되어 있고, 이에 따라 순서에 맞게 결과를 제공할 수 있다.

이론은 이것으로 충분하다. 어떻게 하이버네이트로 자동차 목록을 영속화시킬 수 있을까? 예제를 한번 보자.

4.2.1 예제: 자동차 전시실

간단히 자동차 전시실을 떠올려보자. 각 전시실에는 고객이 보고 구매할 수 있도록 많은 자동차가 전시되어 있다. 이것을 간단한 모델로 표현할 수 있다.

모든 전시실에는 다양한 자동차가 전시된다. 새 차일 수도 중고차일 수도 있다. 전시할 자동차들을 `java.utill.List`로 만든다. 다음은 `Showroom` 클래스의 구현부다.

```
// Showroom 클래스
public class Showroom {
     private int id = 0;
     private String manager = null;
     private String location = null;
     private List<Car> cars = null;
     …
     // Getters와 setters
}

// Car 클래스
public class Car {
     private int id;
     private String name = null;
     private String color = null;
     …
     // Getters와 setters
}
```

Showroom과 Car 클래스는 모두 간단한 POJO 객체다. 여기서 주목할 점은 전시실(Showroom)과 자동차(Car)는 일대다의 연관 관계로 정의된다는 것이다. 한 개(일)의 전시실은 많은(다) 자동차가 있다.

Showroom, Car 등의 POJO 클래스가 준비되었다면 매핑 정의를 추가해 보자.

```
<hibernate-mapping package="com.madhusudhan.jh.collections.list">

    <!-- Showroom  클래스 매핑 선언 -->
    <class name="Showroom" table="SHOWROOM_LIST">
        <id column="SHOWROOM_ID" name="id">
            <generator class="native"/>
        </id>
        …
        <list name="cars" cascade="all" table="CARS_LIST">
            <key column="SHOWROOM_ID"/>
            <index column="CAR_INDEX"/>
            <one-to-many class="Car"/>
        </list>
    </class>

    <! - Car 클래스 매핑 선언 -->
    <class name="Car" table="CARS_LIST">
        <id column="CAR_ID" name="id">
            <generator class="native"/>
        </id>
        <property column="NAME" name="name"/>
        <property column="COLOR" name="color"/>
    </class>
</hibernate-mapping>
```

리스트 요소의 사용에 주목해 보자. 이는 Showroom 객체에 선언된 cars 변수의 테이블 매핑을 정의하는 요소다.

주 테이블인 SHOWROOM_LIST는 예상대로 생성되고 입력된다. 그래서 별로 놀라운 일은 아니지만, 부가적인 테이블인 CARS_LIST는 SHOWROOM_LIST 생성 과정 중에 생성된다. CAR_ID, NAME, COLOR 속성은 CARS_LIST 테이블을 나타내고, Car 객체에 직접 선언된다. 하이버네이트에서는 두 컬럼을 생성한다. 하나는 외래키인 SHOWROOM_ID고 다른 하나는 리스트 인덱스 정보를 담고 있는 CAR_INDEX 컬럼이다. CAR_INDEX에는 각 자동차 리스트 요소들의 위치 정보가 들어간다. 그리고 나중에 원래 위치에서 리스트 요소의 순서를 재구성하는 데 사용된다.

자동차 정보를 조회할 때 하이버네이트에서는 CAR_INDEX의 인덱스 정보에 따라 레코드 순서를 재정렬한다. 이 과정이 어떻게 이루어지는지 알아보기 위해 테스트 프로그램을 실행해 보자.

4.2.2 테스트 프로그램

리스트 영속성 기능 테스트를 위해 간단한 테스트 프로그램을 실행해 보자.

```
private void persistLists() {
    // showroom 객체 생성
    Showroom showroom = new Showroom();
    showroom.setLocation("East Croydon, Greater London");
    showroom.setManager("Barry Larry");

    // cars 리스트 생성
    List<Car> cars = new ArrayList<Car>();
    cars.add(new Car("Toyota", "Racing Green"));
    cars.add(new Car("Toyota", "Racing Green"));
    cars.add(new Car("Nissan", "White"));
```

```
        cars.add(new Car("BMW", "Black"));
        cars.add(new Car("Mercedes", "Silver"));
        ...
        // cars를 showroom에 영속화
        showroom.setCars(cars);

        // showroom 저장
        session.save(showroom);
    }
```

테스트 프로그램 자체가 충분한 설명을 하고 있다. 리스트에 Toyota가 추가되었음을 주목하라. 결과를 조회하기 위해 테스트 프로그램을 실행했을 때 다음과 같은 결과가 콘솔에 출력된다. 결과에 중복된 Toyota car가 있다는 것을 확인할 수 있다. 하이버네이트에서 리스트의 자동차 입력 순서는 보장된다.

```
Showroom{id=6, manager=Barry Larry, location=East Croydon, Greater London,
cars=[
     Car{id=15, name=Toyota, color=Racing Green},
     Car{id=16, name=Toyota, color=Racing Green},
     Car{id=17, name=Nissan, color=White},
     Car{id=18, name=BMW, color=Black},
     Car{id=19, name=Mercedes, color=Silver}]}
```

4.3 Set 영속화

java.util.Set은 중복을 허용하지 않으며, 순서를 보장하지 않는 자료 구조다. 리스트처럼 세트를 이용하는 것은 간단하다. 세트 컬렉션이 하이버네이트에서 어떻게 동작하는지 보기 위해서 자동차 전시실 예제로 다시 돌아가 보자.

변경된 예제에서는 전시실에 있는 자동차 컬렉션을 java.util.Set로 만들고, Set 타입으로 cars 변수를 정의한다. Set 인터페이스의 구현체로는 HashSet을 사용한다.

Showroom 클래스는 다음과 같다.

```
public class Showroom {
     private int id = 0;
     private String manager = null;
     private String location = null;

     // Cars는 세트로 나타냄
     private Set<Car> cars = null;

     // Getters와 setters
     …
```

List 대신 Set 컬렉션을 사용한 것이 눈에 띄는 변화다. Showroom 클래스 변경을 마치면 다음과 같이 set 태그로 Set를 매핑한다.

```
<hibernate-mapping package="com.madhusudhan.jh.collections.set">
     <!-- showroom 클래스. cars의 매핑에 주목하자 -->
     <class name="Showroom" table="SHOWROOM_SET">
        <id column="SHOWROOM_ID" name="id">
           <generator class="native"/>
        </id>
        …
        <set name="cars" table="CARS_SET" cascade="all">
           <key column="SHOWROOM_ID"/>
           <one-to-many class="Car"/>
        </set>
     </class>

     <!-- car 매핑 정의는 매우 간단하다 -->
     <class name="Car" table="CARS_SET">
        <id column="CAR_ID" name="id">
           <generator class="native"/>
```

```
        </id>
        <property column="NAME" name="name"/>
        <property column="COLOR" name="color"/>
    </class>
</hibernate-mapping>
```

Showroom 인스턴스는 SHOWROOM_SET 테이블에 매핑되고, 세트 컬렉션으로 정의된 cars 변수는 CARS_SET 테이블에 매핑된다. key 요소는 CARS_SET 테이블에 외래키가 존재함을 알려준다. 하이버네이트에서는 이 외래키를 자동으로 CAR_SET 테이블에 추가한다. 이런 이유로 하이버네이트에서 생성하고 관리하는 CARS_SET 테이블에는 부가적인 외래키 SHOWROOM_ID가 함께 존재하게 되므로 두 테이블이 연결된다.

다음과 같이 테스트 프로그램을 작성해 보자.

```
private void persistSets() {
    // showroom 을 생성하고 데이터를 덧붙이기
    Showroom showroom = new Showroom();
    showroom.setLocation("East Croydon, Greater London");
    showroom.setManager("Barry Larry");

    // cars 세트를 생성하고 데이터를 덧붙이기
    Set<Car> cars = new HashSet<Car>();
    cars.add(new Car("Toyota", "Racing Green"));
    cars.add(new Car("Nissan", "White"));
    cars.add(new Car("BMW", "Black"));
    cars.add(new Car("BMW", "Black"));

    // cars를 showroom에 연관시키고 영속화하기
    showroom.setCars(cars);
    session.save(showroom);
    …
}
```

앞의 예제에서는 세 개의 새로운 car 인스턴스를 추가한 Showroom 객체를 생성하고, cars 컬렉션 구현체로 HashSet을 사용한다. 여기서 또 하나의 BMW 인스턴스를 추가한 것을 보았는가? 세트 컬렉션에서는 동등성 비교를 기반으로 두 자동차가 같은지 식별하고, 중복되는 것을 제거한다.

세트 컬렉션을 작업할 때 동등성 조건을 충족해야 한다. Car 객체에 equals()와 hashCode() 메소드를 반드시 작성해야 하고, 세트에 추가되는 각 아이템은 고유해야 한다. equals()와 hashCode() 메소드는 이런 조건을 만족하도록 도와준다. equals()와 hashCode()는 유일 조건을 정확히 지켜서 구현되어야 한다. 예를 들어, 한 개의 자동차 변수 car를 고유하게 식별하도록 필드를 이용하면 된다.

retrivieveSets 테스트 메소드는 데이터베이스에서 영속화된 세트를 다음과 같이 조회해서 가져온다.

```
Showroom{id=7, manager=Barry Larry, location=East Croydon, Greater London,
cars=[
     Car{id=27, name=Nissan, color=White},
     Car{id=26, name=Mercedes, color=Silver},
     Car{id=28, name=Toyota, color=Racing Green},
     Car{id=29, name=BMW, color=Black}]}
     …
```

또 다른 BMW 인스턴스를 추가하더라도 목록에서는 BMW가 두 번 나오지 않는데, 이는 세트 컬렉션의 '중복 제거' 정책을 보여준다.

4.4 Map 영속화

이름/값 쌍으로 나타내야 한다면 Map 컬렉션을 택한다. Map 자료 구조는 키(단어)와 관련된 값(의미)을 가진 사전과 같은 구조다. 한 명의 고객(key)과 여러 개의 은행

계좌(value) 또는 주식 발행자를 위한 시세 정보 등 키/값 쌍 형태의 데이터에 맵을 사용한다.

자동차 전시실 예제의 showroom 객체에 잠재 고객의 시험 운전 예약 정보를 넣을 수 있게 추가한다. Map 자료 구조를 사용하고 고객 정보를 자동차 예약 정보에 연결하여 이 기능을 구현한다.

```
public class Showroom {
     private int id = 0;
     private String manager = null;
     private String location = null;
     private Map<String, Car> cars = null;
     // Getters와 setters
     …
```

고객 한 명은 각각의 자동차를 예약할 수 있다. 모든 자동차는 전시실에 속하고, Map<String, Car> 타입으로 고객-자동차 간 자료 타입을 구현한다.

주요 매핑 부분은 다음과 같이 정의된다.

```
<hibernate-mapping package="com.madhusudhan.jh.collections.map">

     <!-- map 태그로 CARS_MAP 테이블에 연결된 cars 변수와 Showroom의 매핑 정의 -->
     <class name="Showroom" table="SHOWROOM_MAP">
        <id column="SHOWROOM_ID" name="id">
           <generator class="native"/>
        </id>
        <property column="MANAGER" name="manager"/>
        …
        <map name="cars" cascade="all" table="CARS_MAP">
           <key column="SHOWROOM_ID"/>
```

```
            <map-key column="CUST_NAME" type="string" />
            <one-to-many class="Car"/>
        </map>
    </class>

    <!- - 간단한 Car 클래스와 테이블 매핑 - - >
    <class name="Car" table="CARS_MAP">
        <id column="CAR_ID" name="id">
            <generator class="native"/>
        </id>
        <property name="name" column="CAR_NAME" />
    <property name="color" column="COLOR" />
    </class>
</hibernate-mapping>
```

Showroom 클래스의 cars 변수는 CARS_MAP 테이블을 참조하는 map 요소로 기술되는데, 여기서 외래키 SHOWROOM_ID는 map 요소에 정의한다. 맵의 키(여기서는 고객)는 map-key 속성으로 정의한다. Car 클래스 매핑은 간단하다. CARS_MAP 테이블에는 SHOWROOM_ID, CUST_NAME 외에 자동차 명(name), 색깔(color) 등 많은 컬럼이 추가된다는 점에 유의하자.

매핑 정의가 끝나면 테스트 프로그램을 통해 어떻게 동작하는지 확인해 보자.

```
private void persistMaps() {
    Showroom showroom = new Showroom();
    showroom.setLocation("East Croydon, Greater London");
    showroom.setManager("Cherry Flurry");

    Map<String, Car> cars = new HashMap<String, Car>();
    cars.put("barry", new Car("Toyota", "Racing Green"));
    cars.put("larry", new Car("Nissan", "White"));
    cars.put("harry", new Car("BMW", "Black"));
```

```
        showroom.setCars(cars);
        ...
    }
```

고객 이름과 시험 주행할 자동차 정보를 가지고 맵을 만들고, 이 맵을 Showroom에 추가한다. 맵 자료 구조에서 볼 수 있듯이, 고객 한 명에 해당하는 새 차 정보를 갖게 된다. 테스트 프로그램의 retrieveMaps 메소드를 실행하면 콘솔에 다음과 같은 결과가 출력된다.

```
Showroom{id=1, manager=Cherry Flurry, location=East Croydon, Greater London,
cars={barry=Car{id=1, name=Toyota, color=Racing Green},
harry=Car{id=2, name=BMW, color=Black},
larry=Car{id=3, name=Nissan, color=White},
fairy=Car{id=4, name=Mercedes, color=Pink}}}
...
```

결과에는 고객과의 시험 주행이 예약된 전시실의 모든 자동차가 출력된다.

4.5 Array 영속화

배열Array 영속화는 리스트 영속화와 비슷하므로 메커니즘에 대해 간단히 넘어가고 자세한 내용은 생략하겠다. Showroom 클래스는 다음과 같이 String 배열 타입의 cars 변수를 가지고 있다.

```
public class Showroom {
    private int id = 0;
    private String manager = null;
    private String location = null;
    // cars의 리스트
    private String[] cars = null;
...
```

클래스 간의 매핑 정보는 다음과 같이 정의된다.

```
<hibernate-mapping package="com.madhusudhan.jh.collections.array">
    <class name="Showroom" table="SHOWROOM_ARRAY">
        <id column="SHOWROOM_ID" name="id">
            <generator class="native"/>
        </id>
        ...
        <array name="cars" cascade="all" table="CARS_ARRAY">
            <key column="SHOWROOM_ID"/>
            <index column="CAR_INDEX"/>
            <element column="CAR_NAME" type="string" not-null="true"/>
        </array>
    </class>
</hibernate-mapping>
```

array 태그에서 cars 변수와 CARS_ARRAY 테이블의 매핑 정보를 정의한다. 역시 CARS_ARRAY 테이블에는 외래키가 존재한다(여기서는 SHOWROOM_ID가 해당). 하이버네이트에서는 입력 순서를 지키기 때문에 index 태그는 반드시 이름과 함께 정의해야 한다. element 태그에는 배열에 들어갈 실제 값을 정의한다. 이 경우에는 각 자동차 모델명이 된다.

다음 테스트 클래스의 persistArrays 메소드는 앞에서 언급한 배열 컬렉션을 영속화한다. 다음과 같이 String[]로 cars를 생성하고 자동차 모델명을 넘겨준다.

```
private void persistArrays() {
    ...
    Showroom showroom = new Showroom();
    showroom.setLocation("East Croydon, Greater London");
    showroom.setManager("Barry Larry");
```

```
        // cars 배열 생성과 showroom과 연관시키기
        String[] cars = new String[]{"Toyota","BMW","Citroen"};
        showroom.setCars(cars);

        // showroom 저장하기
        session.save(showroom);
        …
    }
```

테스트 프로그램의 retrieveArrays 메소드는 다음과 같은 자동차 정보를 조회해서 가져온다.

```
Showroom{id=9, manager=Barry Larry, location=East Croydon, Greater London,
cars=[Toyota, BMW, Citroen]}
```

4.6 Bags와 IdBags 영속화

비순차 컬렉션unordered collection과 인덱스가 없는 요소를 원할 수 있지만 자바에서는 이런 자료 구조를 지원하지 않는다. 가장 유사한 것이 java.util.List지만, 순차적order이며 인덱스를 가지고 있다. 이것을 해결하기 위해서 하이버네이트에서는 백bag이라고 하는 별도의 컬렉션 타입을 지원한다.

백은 리스트와 대조적이다. 비순차적이고 인덱스를 가지고 있지 않으며 중복이 허용되는 컬렉션이다. 백은 하이버네이트에만 있는 자료 구조로 자바에는 동일한 자료 구조가 없다.

백의 구현은 매우 간단하며, 일반 엔티티와 다른 점은 없다. 사실 백을 표현하기 위해 자바 코드에서는 여전히 List를 사용하는데(자바에는 백 컬렉션이 없다), 다른 점은 매핑 부분에서 발생한다. 컬렉션을 list로 선언하는 대신 bag을 사용한다.

엔티티 클래스의 변경이 없는, 다음 매핑 정의를 살펴보자.

```
<hibernate-mapping package="com.madhusudhan.jh.collections.bags">
     <class name="Showroom" table="SHOWROOM_BAGS">
        <id column="SHOWROOM_ID" name="id">
           <generator class="native"/>
        </id>
        …
               <bag name="cars" cascade="all" table="CARS_LIST">
           <key column="SHOWROOM_ID"/>
           <one-to-many class="Car"/>
        </bag>
     </class>
     <class name="Car" table="CARS_BAGS">
        …
  </class>
</hibernate-mapping>
```

bag 요소에서 list 정의에 반드시 존재하는 인덱스 요소가 제거된 것을 눈치챘는가? 백 요소에서는 컬렉션의 인덱스 정보가 더는 영속화되지 않는다. 그래서 백 매핑 정의에서 index 요소는 불필요하다.

이 차이점 외에는 다른 모든 부분은 동일하다. 다음은 정의된 테스트 클래스에서 엔티티 영속화 전에 어떻게 데이터가 들어가는지 보여준다.

```
private void persist() {
     …
     Showroom showroom = new Showroom();
     showroom.setLocation("East Croydon, Greater London");
     showroom.setManager("Barry Larry");
```

```
        // cars 배열 정의하기 - 타입에 주의하자
        String[] cars = new String[]{"Toyota","BMW","Citroen"};

        // showroom변수에 cars변수를 넣고 영속하기
        showroom.setCars(cars);
        session.save(showroom);
        …
}
```

bag은 표준 컬렉션이 아니라 하이버네이트에서 제공하는 명세다. bag을 사용하기 위해 코드에서 여전히 `java.util.List`를 사용하더라도 매핑 정의는 정확히 명시해야 한다. 가능하다면 bag 컬렉션을 사용하지 말고, 표준 컬렉션을 사용하기를 바란다.

bag 외에도 하이버네이트에서는 idbag를 지원한다. 이 컬렉션은 키가 없는 bag과 달리 영속화된 컬렉션 자체에서 대리키를 갖는 메커니즘을 제공한다. 보통 POJO가 변경되지 않으므로 매핑 정보에 주의한다.

```
<hibernate-mapping package="com.madhusudhan.jh.collections.idbags">
    <class name="Showroom" table="SHOWROOM_IDBAGS">
        <id column="SHOWROOM_ID" name="id">
            <generator class="native"/>
        </id>
        …
                <idbag name="cars" cascade="all" table="SHOWROOM_CARS_IDBAGS">
            <collection-id  column="SHOWROOM_CAR_ID" type="long">
                <generator class="hilo"/>
            </collection-id>
            <key column="SHOWROOM_ID"/>
            <many-to-many class="Car" column="CAR_ID"/>
```

```
        </idbag>
    </class>
    <class name="Car" table="CARS_IDBAGS">
        <id column="CAR_ID" name="id">
            <generator class="native"/>
        </id>
        ...
    </class>
</hibernate-mapping>
```

cars 컬렉션을 사용하기 위해 조인 테이블 SHOWROOM_CARS_IDBAGS를 참조하는 idbags 요소를 도입한다. collection-id 요소는 조인 테이블의 기본키를 생성하고, 조인 테이블이 가진 기본키 외에도 참조하는 두 테이블의 기본키 또한 가지고 있다.

idbag 컬렉션은 자주 사용하지 않으므로 idbag을 사용해야 한다면 한 번 더 검토해 보길 바란다.

4.7 어노테이션을 이용한 컬렉션 영속화

앞에서 XML 매핑 방법을 이용하여 컬렉션을 저장하는 내부 동작을 살펴보았는데, 어노테이션을 이용해서도 컬렉션을 영속화할 수 있다. 우선 할 일은 적절한 어노테이션을 엔티티 클래스에 기록하는 것이다. 여기서는 자동차 전시실 예제를 개선해 보자.

어노테이션을 이용하는 방법은 두 가지가 있다. 외래키를 사용하는 방법과 중간에 조인 테이블을 이용한 방법이다. 두 가지 방법을 자세히 알아보자.

4.7.1 외래키 이용하기

각 전시실에는 많은 자동차가 있다. 이는 일대다 관계를 의미한다. Showroom 엔티티 클래스는 자동차 컬렉션으로 이루어져 있고, 고객에게 전시된다. 자동차는 쇼케이스에 속하고 전시실 외래키를 통해 자동차 전시실과 서로 연결되어야 한다.

Showroom 엔티티 클래스부터 살펴보자.

```
@Entity(name="SHOWROOM_LIST_ANN")
@Table(name="SHOWROM_LIST_ANN")
public class Showroom {
     @Id
     @Column(name="SHOWROOM_ID")
     @GeneratedValue(strategy=GenerationType.AUTO)
     private int id = 0;

     @OneToMany
     @JoinColumn(name="SHOWROOM_ID")
     @Cascade(CascadeType.ALL)
     private List<Car> cars = null;

     // 다른 속성들
     private String manager = null;
     private String location = null;
```

클래스는 @Entity 어노테이션을 통해서 영속화할 수 있는 엔티티로 선언된다. 그리고 @Table 어노테이션의 테이블과 매핑되어 있다. 자동 생성 방법을 이용하여 식별자를 선언한다. 이는 auto_increment나 identity 같은 데이터베이스에서 제공하는 함수의 하나를 식별자 방법으로 사용한다는 의미다.

전시실의 중요 속성에 초점을 맞춰보자. cars 변수는 자동차 컬렉션으로 사용된다. 자동차 정보를 담기 위해 java.util.List 컬렉션을 사용한다. 이 변수는 @

OneToMany 어노테이션과 함께 기술된다. 각 전시실은 다수의 자동차를 갖고 있고, 각 자동차는 한 전시실에 속하기 때문이다.

앞에서 cars 컬렉션이 showroom 테이블의 기본키를 참조하는 외래키를 가진 테이블이라는 점을 언급했다(이 경우에는 SHOWROOM_ID).

이와 같은 테이블 간 의존성을 하이버네이트에 제공하기 위해서 cars 변수에 외래키가 정의된 @JoinColumn 어노테이션과 함께 명시해야 한다. cars 테이블에서 자동차 리스트를 뽑아오기 위해서 반드시 컬럼명 SHOWROOM_ID를 제공해야 한다. @Cascade는 주 인스턴스인 showroom 인스턴스와 함께 cars 컬렉션 영속화가 가능하게 한다.

Car 엔티티 클래스는 간단하고 단순하다.

```
@Entity(name="CAR_LIST_ANN")
@Table(name="CAR_LIST_ANN")
public class Car {
     @Id
     @GeneratedValue(strategy= GenerationType.AUTO)
     @Column(name="CAR_ID")
     private int id;
     private String name = null;
     private String color = null;
     …
}
```

showroom과 cars 데이터베이스 테이블 스크립트는 다음과 같다.

```
-- showroom를 위한 테이블
CREATE TABLE showroom_list_ann (
```

```
    SHOWROOM_ID int(10) NOT NULL AUTO_INCREMENT,
    location varchar(255) DEFAULT NULL,
    manager varchar(255) DEFAULT NULL,
    PRIMARY KEY (SHOWROOM_ID)
)
-- cars를 위한 테이블
CREATE TABLE car_list_ann (
    CAR_ID int(11) NOT NULL AUTO_INCREMENT,
    color varchar(255) DEFAULT NULL,
    name varchar(255) DEFAULT NULL,
    SHOWROOM_ID int(11) DEFAULT NULL,
    PRIMARY KEY (CAR_ID),
    FOREIGN KEY (SHOWROOM_ID) REFERENCES showroom_list_ann (SHOWROOM_ID)
)
```

cars 테이블 내에는 기본키(CAR_ID)와 showroom 테이블을 참조하는 외래키(SHOWROOM_ID)가 있다.

다음과 같이 Configuration 클래스에 어노테이션 클래스를 추가하여 테스트 케이스를 준비한다.

```
Configuration config = new Configuration()
    .configure("collections/list/ann/hibernate.cfg.xml")
    .addAnnotatedClass(Showroom.class)
    .addAnnotatedClass(Car.class);
```

테스트 프로그램의 영속화 메커니즘은 앞에서 살펴본 리스트 영속화 과정과 다르지 않으므로 테스트 케이스를 반복해서 언급하지 않겠다.

showroom의 결과는 다음과 같다.

```
Showroom{id=1, manager=Barry Larry, location=East Croydon, Greater London,
cars=[
     Car{id=1, name=Toyota, color=Racing Green},
     Car{id=2, name=Nissan, color=White},
     Car{id=3, name=BMW, color=Black},
     Car{id=4, name=Mercedes, color=Silver}]}
```

여기까지 외래키를 이용한 컬렉션 영속화 방법을 살펴보았다. 조인 테이블을 이용한 두 번째 방법을 알아보자.

4.7.2 조인 테이블 이용하기

조인 테이블 전략을 이용할 때 양 테이블의 기본키를 가지고 연계하는 매핑(조인) 테이블이 있어야 한다. 예를 들어, 다음 조인 테이블은 showroom과 cars 두 테이블의 기본키로 이루어져 있다.

```
// showroom과 cars를 위한 조인 테이블
SHOWROOM_ID     CAR_ID
1               1
1               2
2               1
2               2
```

다음 사항을 Showroom 엔티티 클래스에 반영하고, cars 테이블을 알맞게 명시해 보자.

```
@Entity(name="SHOWROOM_SET_ANN_JOINTABLE")
@Table(name="SHOWROOM_SET_ANN_JOINTABLE")
public class Showroom {
     @Id
```

```
    @Column(name="SHOWROOM_ID")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id = 0;
    private String manager = null;
    private String location = null;

    @OneToMany
    @JoinTable
    (name="SHOWROOM_CAR_SET_ANN_JOINTABLE",
        joinColumns = @JoinColumn(name="SHOWROOM_ID")
    )
    @Cascade(CascadeType.ALL)
    private Set<Car> cars = null;
 …
}
```

앞에서 본 @joinTable 어노테이션 부분에서는 SHOWROOM_CAR_SET_ANN_JOINTABLE 연계 테이블을 이용하겠다고 알려준다. SHOWROOM_ID 조인 컬럼을 통해 자동차 정보를 조회하는 것에 주의한다.

생성된 조인 테이블은 다음과 같다.

```
CREATE TABLE showroom_car_set_ann_jointable (
    SHOWROOM_ID int(11) NOT NULL,
    car_id int(11) NOT NULL,
    PRIMARY KEY (SHOWROOM_ID,car_id),
    FOREIGN KEY (SHOWROOM_ID) REFERENCES showroom_set_ann_jointable (SHOWROOM_ID),
    FOREIGN KEY (car_id) REFERENCES car_set_ann_jointable (id)
)
```

기본키는 showroom_id와 car_id의 조합이다. 또한 전시실과 자동차의 개별 테이블로 외래키를 정의한다는 점에 주의하자.

4.8 요약

이번 장에서는 하이버네이트 프레임워크로 컬렉션을 영속화하는 다양한 방법을 살펴보았다. 그리고 엔티티 클래스와 매핑 정의를 자세히 살펴보았다. 리스트, 세트, 배열, 백, 맵 관련 예제들을 살펴보고, 어노테이션을 이용한 컬렉션 영속화 방법을 알아보았다.

5 | 연관 관계

객체 영속화 세상에서는 연관 관계association와 관계relationship에 대한 이해는 필수다. 복잡성을 다루지 못한다면 현실 문제를 해결하기 힘들다. 이 장에서는 연관 관계의 개념과 매핑 방법을 집중적으로 살펴보고, 다양한 연관 관계의 방향성directionality과 다중성multiplicity을 알아보겠다.

삶에는 언제나 단순한 상황만 있는 것이 아니듯, 객체는 객체 자신만으로는 오래 유지될 수가 없다. 현실 문제와 해결 방법이 반영된 다른 객체와의 관계가 필요하다. 지금까지 Movie, Trade와 같은 단순한 자바 객체의 영속화 방법과 접근 방법을 살펴보았다. 줄거리, 대사, 배우, 기술자와 같은 콘텐츠가 없는 영화는 거의 없듯이 Trade도 기본적인 상대 계약자, 수량, 관련 정보 등과 항상 연관 관계가 있다.

이러한 연관 관계는 중요한 역할을 한다. 그리고 관계형 데이터베이스에서 데이터를 조회하거나 영속화할 때 연관 관계를 이해해야 한다. ORM 도구로 이것을 반영하는 것은 매우 복잡한 작업이다. 하이버네이트에서는 이를 쉽게 할 수 있도록 기능을 제공하고 있다. 하이버네이트의 진정한 힘은 객체 간의 연관 관계와 테이블 간의 관계를 관리할 수 있도록 제공하는 기술에 있다.

5.1 연관 관계

자바 객체 간의 연관 관계를 이해하는 것은 하이버네이트 작업에서 매우 중요하다. 자바에서 연관 관계를 표현하는 것은 간단히 클래스의 속성(변수)을 이용하면 된다. 이미 연관 관계와 관련된 작업을 한 적이 있다. 다음의 두 클래스(Car, Engine)의 정의 부분을 확인해 보자.

```
// Car 클래스
public class Car {
	private int id;
	// Car는 engine을 가지고 있다.
	private Engine engine;
}

// Engine 클래스
public class Engine {
	private int id = 0;
	private String make = null;
	…
}
```

Car POJO 안에 engine 속성이 함께 정의된다. 앞의 코드는 Car 클래스가 Engine 클래스와 연관 관계가 있다는 것을 보여준다. 자바에서는 객체 간의 연관 관계를 만들기 위해 속성을 사용한다.

다음으로 넘어가기 전에 용어를 이해해 보자. 테이블 간의 관계는 관계형 데이터베이스 테이블 간의 연관 관계다. 이와 같은 테이블 사이의 관계는 대부분 기본키, 외래키 제약 조건을 통해서 표현된다. 연관 관계와 관계라는 용어를 종종 혼용해서 쓰지만, 객체에서는 연관 관계를 사용하고 테이블에서는 관계를 사용하겠다.

연관 관계에서 반드시 기억할 두 가지는 다중성multiplicity과 방향성directionality이다.

5.1.1 다중성

다중성이란 얼마만큼의 특정 객체가 얼마만큼의 대상 객체와 연관 관계가 있는지를 말한다. 은행 계좌는 한 개나 여러 개 또는 아예 가지고 있지 않을 수도 있다. 이웃 에는 아이가 한 명이나 여러 명 또는 전혀 없을 수도 있다.

객체 간의 연관 관계나 테이블 간의 관계를 다룰 때마다 다중성의 시각으로 그 관계를 고려해야 한다. 모든 자동차는 한 개의 엔진이 있고, 각 엔진은 한 대의 차에 속한다. 쉽게 말해, 자동차 한 대(일)는 엔진 한 개(일)를 갖는다. 그러므로 여기에서 다중성은 일대일 연관 관계로 표현된다.

여러 명의 배우가 참여하는 영화가 있다. 한(one)의 영화에는 한(one) 명 또는 다수(more/many)의 배우가 있다. 이 다중성을 일대다(one-to-many)로 표현한다(배우들은 다른 영화에서 연기할 수도 있어서 다대다로 만들어야 하지만 이 예제에서는 이를 무시한다).

마지막 다중성 연관 관계는 다대다(many-to-many)다. 한 명의 학생은 많은(many, 0 또는 그 이상) 수업을 등록할 수 있고, 각 수업은 많은(many) 학생들로 구성된다. 이 문맥에서 많은(many)은 영, 일, 다(큰 수일 필요는 없다)를 의미한다.

[표 5-1]에서 보듯이 기본적으로 세 종류의 연관 관계가 있다.

[표 5-1] 연관 관계의 종류

종류	정의	예
일대일	한 테이블에서 각 레코드는 반드시 다른 테이블의 레코드 한 개와 관계가 있다. 반대의 경우도 마찬가지다. 다른 테이블의 레코드는 0일 수도 있다.	자동차 한 대는 오직 한 개의 엔진만 가진다.
일대다 또는 다대일	한 테이블에서 각 레코드는 다른 테이블의 0개 또는 그 이상의 레코드와 관계가 있다.	영화 한 편은 많은 배우를 가진다(일대다). 배우 한 명은 여러 작품에서 연기할 수 있다(다대일).
다대다	양쪽 테이블 모두 각 레코드가 다른 쪽 테이블의 0개 또는 그 이상의 레코드와 관계가 있다.	

5.1.2 방향성

연관 관계의 또 다른 속성은 방향성이다. 이 속성은 연관 관계의 방향을 정의한다. 예를 들어, `Car`와 `Engine`의 관계에서 `Car`의 속성을 질의해서 `Engine`을 찾아낼 수

있다. 즉, Car 객체를 제공하면 어떤 엔진을 가지고 있는지 알 수 있다. 다른 예로, 대학에 재학 중인 학생을 살펴보자. Student 객체의 course 속성을 질의하여 무슨 수업에 등록했는지 알 수 있다.

연관 관계는 단방향일 수도 있고, 양방향일 수도 있다. 자동차와 엔진 예제에서 자동차 정보를 제공받으면 무슨 엔진인지 알 수 있다. 그러나 엔진의 상세 정보를 제공받는다면 자동차 모델명을 알 수 있을까? 안타깝게도 엔진 정보로 자동차의 세부 정보는 알 수는 없다. 이와 같은 연관 관계는 방향성이 한 쪽에만 존재하는 단방향 타입이다. 자바에서는 다른 방법이 아닌 임의의 클래스에서 참조할 대상 클래스를 생성한다. 앞에서 살펴본 Car와 Engine에서 Car 클래스가 engine 속성을 이용하여 연관 관계를 가진 것을 보았다. 그러나 Engine 클래스는 Car를 참조할 속성이 없다(그래서 자동차 정보를 얻지 못한다).

임의의 객체에서 대상 객체를 찾거나 반대의 경우에도 같다면 그 관계를 양방향이라 한다. Car 클래스와 Owner 클래스의 경우 주어진 Car 객체로 자동차의 주인이 누구인지 알 수 있으며, Owner 객체로 차주의 자동차가 무엇인지 알 수 있다. 다음의 Owner 클래스와 Car 클래스 정의를 살펴보자.

```
// Owner 클래스
public class Owner {
	private int id = 0;
	// 차주가 소유한 자동차의 목록을 얻을 수 있다.
	// 그러나 단순하게 설명하기 위해 한 대의 자동차로 하자.
	private Car car = null;
	…
}

// Car 클래스
public class Car {
```

```
        private int id = 0;
        // 차주
        private Owner owner = null;
        …
}
```

양방향성 연관 관계를 유지할 수 있도록 Owner 객체에 Car에 대한 참조를 제공하고, Car 객체에는 Owner에 대한 참조를 제공하고 있다.

앞의 예제는 일대일 양방향 연관 관계다. 이 방향성에 다중성을 추가해 보자.

다대다 양방향 연관 관계는 어떨까? 학생과 강의 예제는 이 같은 연관 관계를 보여준다. Student 객체로부터 한 명의 학생이 등록한 강의들을 알 수 있다. 동시에 Course 객체를 질의하여 특정 수업을 등록한 학생 명단을 알아낼 수 있다. 이 내용은 다음의 Student와 Cource 클래스에 정의되어 있다.

```
// Student 클래스
public class Student {
        private int id = 0;
        // 학생이 등록한 강의 목록
        private List<Course> courses = null;
        …
}

// Course 클래스
public class Course {
        private int id = 0;
        // 강의를 등록한 학생 명단
        private List<Student> students = null;
        …
}
```

두 클래스 모두 다른 쪽에 대한 참조가 있고, 콜렉션 변수라는 것을 눈치챘는가? 이는 양방향이고 다대다 연관 관계를 표현하는 방법이다.

여기까지 연관에 관한 기본 개념을 살펴보았다. 자바에서 이것을 어떻게 구현하는지 차근차근 살펴보자.

5.2 일대일 연관 관계

Car와 Engine 클래스 예제로 일대일 양방향 연관 관계를 만들어보자. 다시 정리하면, 모든 자동차는 엔진 한 개를 가진다. 그리고 모든 엔진은 자동차 한 대에 장착된다. 그래서 일대일 매핑을 나타낸다.

일대일 연관 관계를 설정하는 방법은 두 가지다. 기본키를 사용하는 방법과 외래키를 사용하는 방법이다. 객체 모델에서는 그 차이점이 두드러지지 않지만, 관계형 모델에서는 분명한 차이를 보인다. 이 두 가지 모델에 대해서 살펴보는데, 두 모델의 영속화 엔티티 클래스는 동일하다. 우선 클래스 선언부터 시작해 보자.

다음은 Car와 Engine 클래스의 선언부다.

```
// Car 클래스
public class Car {
      private int id;
      private String name;
      private String color;
      // 차는 하나의 엔진을 가진다.
      private Engine engine;
      …
}

// Engine 클래스
public class Engine {
```

```
        private int id = 0;
        private String make = null;
        private String model = null;
        private String size = null;
        // 이 엔진은 하나의 차에 장착되어 있다.
        private Car car = null;
}
```

Engine 클래스의 Car와 Car 클래스의 Engine이 engine과 car 변수로 참조된다. 이는 자바에서 연관 관계를 표현하는 방법이다.

이제 테이블 스키마를 알아보자.

일대일 관계를 표현하는 데 기본키 또는 외래키를 이용하는 방법이 있다. 두 방법 모두 자바 클래스는 동일하게 유지되지만, 테이블 정의는 달라진다.

5.2.1 기본키 사용하기

동일한 기본키를 공유하여 두 테이블의 일대일 관계를 나타낸다. 테이블은 다음과 같이 기본키를 공유하도록 설계된다.

```
// CAR 테이블
CREATE TABLE CAR (
       CAR_ID int(10) NOT NULL,
       NAME varchar(20) DEFAULT NULL,
       COLOR varchar(20) DEFAULT NULL,
       PRIMARY KEY (CAR_ID))

// ENGINE 테이블
CREATE TABLE ENGINE (
       CAR_ID int(10) NOT NULL,
       SIZE varchar(20) DEFAULT NULL,
```

```
    MAKE varchar(20) DEFAULT NULL,
    MODEL varchar(20) DEFAULT NULL,
    PRIMARY KEY (CAR_ID),
    FOREIGN KEY (CAR_ID) REFERENCES car (CAR_ID))
```

CAR 테이블은 기본키로 CAR_ID를 갖는 간단한 구조다. 그러나 흥미로운 부분은 ENGINE 테이블에 있다. 여기서 짚고 넘어갈 두 가지 사항이 있다.

- Engine 테이블의 기본키는 CAR_ID이다.
- CAR 테이블의 기본키를 가리키는 외래키 제약 조건이 있다. 그래서 엔진은 자동차와 동일한 id로 생성된다. 따라서 두 테이블 모두 동일한 기본키를 공유한다고 말할 수 있다.

이제 일대일 연관 관계를 어떻게 매핑하는지 살펴보자. CAR 테이블에 상응하는 Car 객체의 매핑 방법은 다음의 Car.hbm.xml 파일과 같다

```
<hibernate-mapping package="com.madhusudhan.jh.associations.one2one">
      <class name="Car" table="CAR">
          <id name="id" column="CAR_ID">
              <generator class="assigned"/>
          </id>
          <property name="name" column="NAME"/>
          <property name="color" column="COLOR"/>
          <one-to-one name="engine" class="Engine" cascade="all"/>
      </class>
</hibernate-mapping>
```

id 값은 애플리케이션을 통해서 정해진다. 그리고 관련 속성들은 테이블의 컬럼에 연결된다. Car 클래스와 engine 속성을 연결하기 위해 one-to-one 매핑 태그가 사용된다. 다음은 이 표기법에 대한 설명이다.

- Car 인스턴스(Car 클래스를 위해 정의)는 engine이라는 속성을 가진다.
- Car와 Engine 클래스는 일대일 연관 관계다.
- Engine 값은 ENGINE 테이블(Engine 객체와 매핑)의 값으로 정해진다.

Engine 매핑 안에는 더 많은 것이 포함되어 있다.

```
<hibernate-mapping package="com.madhusudhan.jh.associations.one2one">
      <class name="Engine" table="ENGINE">
      <id name="id" column="CAR_ID" >
          <generator class="foreign">
              <param name="property">car</param>
          </generator>
      </id>
      <one-to-one name="car" class="Car" constrained="true"/>
      <property name="size" column="SIZE"/>
       ...
      </class>
</hibernate-mapping>
```

Engine의 기본키(id)가 Car의 id와 동일하다. 그리고 이런 상황을 하이버네이트에서 알 수 있도록 매핑할 때 관련 내용을 명시해야 한다. 이런 목적으로 foreign이라는 특별한 식별자 생성 클래스가 있다. 이 생성 클래스는 car로 명시된 속성 정보가 있는지 확인한다(car는 매핑 하단에 one-to-one 태그를 통해 정의한다). 그리고 그 참조로부터 id 정보를 가져온다.

one-to-one 요소에는 constrained="true"라는 속성이 한 가지 더 사용된다. 즉, ENGINE 테이블의 기본키는 외래키 제약 조건이 있다는 의미다. 이 외래키 정보는 CAR 테이블의 기본키로부터 알 수 있다.

어려운 작업은 다 했다. 이제 프로그램을 실행해 보자.

5.2.2 연관 관계 테스트하기

테이블 간의 일대일 연관 관계가 어떻게 동작하는지 알아보기 위해 다음과 같이 테스트 클래스를 작성해 보자.

```
/* 기본키를 공유한 일대일 매핑 테스트*/
public class OneToOneTest {
        …
        private void persist() {
            …
            // 우선 Car 인스턴스를 만들고, id와 다른 속성을 설정한다.
            Car car = new Car();

            // id 생성을 위해 애플리케이션 식별자 생성자를 사용한다.
            car.setId(1);
            car.setName("Cadillac ATS Sedan");
            car.setColor("White");

            // 다음으로 engine 인스턴스를 생성하고, 값을 설정한다.
            // 주의: id 값을 지정하지 않는다.
            Engine engine = new Engine();
            engine.setMake("V8 Series");
            engine.setModel("DTS");
            engine.setSize("1.6 V8 GAS");

            // 이제 car에서 setter를 사용해서 연관시킨다.
            car.setEngine(engine);
            engine.setCar(car);

            // 끝으로, 영속화시킨다.
            session.save(car);
            session.save(engine);
            …
        }
}
```

소스 코드가 많지만, 파악하는 데 큰 어려움이 없을 것이다. Car 객체로부터 id 정보를 빌려서 공유하므로 Engine 객체를 생성하는 동안에는 Engine의 기본키 값을 지정하지 않는다는 점에 주의하자. Engine 객체가 생성되는 과정 중에 하이버네이트는 Car로부터 id 정보를 얻어 Engine 객체에 넣는다.

앞의 소스 코드에서 car와 engine 인스턴스를 모두 영속화한다. 기본키-외래키 제약 조건만 제외하고 두 테이블은 거의 별개의 테이블이다.이어서 car 인스턴스의 영속화만으로도 engine 인스턴스를 영속화할 수 있는 방법을 살펴보자.

[표 5-2]와 [표 5-3]은 테이블의 결과다.

[표 5-2] Car 테이블

CAR_ID	COLOR	NAME
1	White	Cadillac ATS Sedan

[표 5-3] Engine 테이블

CAR_ID	SIZE	MAKE	MODEL
1	1.6	V8 Gas	V8

두 테이블에 모두 공유된 기본키 CAR_ID를 볼 수 있다.

5.2.3 외래키 이용하기

외래키 전략을 사용하려면 테이블과 매핑 정의 변경이 필요하다(POJO는 동일하게 유지한다). 변경된 테이블 선언부는 다음과 같다(앞의 자동차 예제와 겹치는 부분을 방지하고자 테이블명에 접미사 V2를 붙여 변경했다).

```
CREATE TABLE CAR_V2 (
       CAR_ID int(10) NOT NULL,
       ENGINE_ID int(10) NOT NULL,
```

```
        COLOR varchar(20) DEFAULT NULL,
        NAME varchar(20) DEFAULT NULL,
        PRIMARY KEY (CAR_ID),
        CONSTRAINT FK_ENG_ID FOREIGN KEY (engine_id) REFERENCES ENGINE_v2 (ENGINE_ID)
)

CREATE TABLE ENGINE_V2 (
        ENGINE_ID int(10) NOT NULL,
        MAKE varchar(20) DEFAULT NULL,
        MODEL varchar(20) DEFAULT NULL,
        SIZE varchar(20) DEFAULT NULL,
        PRIMARY KEY(ENGINE_ID)
)
```

ENGINE_V2 테이블 정의는 간단하고 이해하기 쉽다. ENGINE_ID를 기본키로 하는 일반적인 테이블이다. 눈에 띄는 변경 사항은 CAR_V2 테이블에 있다. 기본키(CAR_ID)뿐만 아니라 ENGINE_V2를 바라보는 외래키(ENGINE_ID)도 정의되어 있다.

다음은 매핑 정의 단계다. Engine 매핑 정의는 특별한 것이 없다.

```
<hibernate-mapping package="com.madhusudhan.jh.associations.one2one">
        <!-- 외래키 사용-->
        <class name="Engine" table="ENGINE_V2">
        <id name="id" column="ENGINE_ID" >
            <generator class="assigned"/>
        </id>
        <property name="size" column="SIZE" />
        …
        </class>
</hibernate-mapping>
```

객체 id를 정하기 위해 애플리케이션 방법을 사용하고, 클래스 속성과 연관 관계인

테이블 컬럼을 매핑한다. 다음은 Car 객체를 위한 매핑 정의 부분이다.

```
<hibernate-mapping package="com.madhusudhan.jh.associations.one2one">
      <class name="Car" table="CAR_V2">
          <id name="id" column="CAR_ID" >
              <generator class="assigned"/>
          </id>
          <property name="name" column="NAME" />
          <property  name="color" column="COLOR"/>
          <!—unique="true"는 일대일로 다대일 관계를 만든다.
              이것은 이 방법을 위한 아주 중요한 설정이다. -->
          <many-to-one name="engine" class="Engine" column="engine_id"
unique="true" cascade="all" />
      </class>
</hibernate-mapping>
```

앞에서 사용한 one-to-one 요소 대신 "unique=true" 속성이 추가된 many-to-one 요소를 사용한다는 점에 주의하자. 이 속성으로 다대일 관계를 일대일 연관 관계로 만들 수 있다. 그리고 Car 엔티티와 Engine 엔티티가 일대일 연관 관계가 되도록 보장한다.

변경 부분을 반영하기 위해 테스트 프로그램을 수정해 보자.

```
private void persistV2() {
      // Engine 생성하기
      Engine e = new Engine();
      e.setId(1);
      e.setMake("V8 Series");
      e.setModel("DTS");
      e.setSize("1.6 V8 GAS");
```

```
        // Car 생성하기
        Car car = new Car();
        car.setId(1);
        car.setName("Cadillac ATS Sedan");
        car.setColor("White");

        // 둘을 연관시키기
        car.setEngine(e);

        // save() 메소드를 사용하여 car를 영속화한다.
        // 주의: Engine은 매핑 안에서 정의한 연속된 속성이므로 자동으로 저장된다.
        session.save(car);
        ...
}
```

앞의 코드에서 엔티티 클래스 Engine과 Car를 각각 생성하는 것을 살펴보았다. Car 엔티티에 Engine 엔티티를 추가하고 데이터베이스에 저장했다. Engine 엔티티를 명확하게 저장하지 않는다. Car를 영속화하는 동안 연관 관계된 객체(여기서는 Engine) 역시 저장된다. 매핑 파일에 cascade="all" 속성이 정의되었기 때문이다.

일대일 연관 관계에 대한 설명을 마치기 전에 어떤 방법이 더 좋은 방법인지 궁금할 것이다. 개인적으로는 unique 속성이 true인 many-to-one 요소를 이용한 외래키 관계를 선호한다. 하이버네이트에서도 이 방법을 권장한다.

5.2.4 어노테이션

어노테이션을 사용할 때는 POJO에 어노테이션을 표기해야 한다.

```
@Entity
@Table(name="CAR_ONE2ONE_ANN")
public class Car {
```

```
    @Id
    @Column(name="CAR_ID")
    @GeneratedValue(strategy= GenerationType.AUTO)
    private int id;
    private String name = null;
    @OneToOne (cascade= CascadeType.ALL)
    @JoinColumn(name="ENGINE_ID")
    private Engine engine = null;
    …
```

Car 클래스는 영속화 대상을 만들기 위해 @Entity 어노테이션을 표기한다. @Id 어노테이션은 테이블의 CAR_ID를 가리키는 id 필드에 선언되고, @OneToOne 어노테이션을 사용해서 일대일 매핑을 정의한다. Car 클래스와 ENGINE 테이블을 조인하기 위해서 ENGINE_ID 컬럼에 @JoinColumn을 사용한다.

Engine 클래스는 아주 간단하다. @Entity, @Table 어노테이션 외에는 일반적인 클래스의 레벨 정의다. 연관 관계 어노테이션 @OneToOne은 car 필드에 명시한다.

```
@Entity
@Table(name="ENGINE_ONE2ONE_ANN")
public class Engine {
      @Id
      @Column(name="ENGINE_ID")
      @GeneratedValue(strategy= GenerationType.AUTO)
      private int id = 0;
      @OneToOne(mappedBy="car")
      private Car car = null;
      …
```

세션 팩토리를 초기화하는 동안 두 AJO를 추가하고 기능을 테스트하자.

지금까지 실제 돌아가는 일대일 연관 관계를 살펴보았다. 다음 절에서는 다대일과 일대다 연관 관계를 살펴보겠다.

이미 일대일 연관 관계를 자세히 다루었다. 이 개념은 모든 연관 관계에 적용할 수 있으므로 다른 연관 관계는 자세히 다루지는 않고 연관 관계 사이의 차이점을 중점으로 살펴보겠다.

5.3 일대다 또는 다대일 연관 관계

영화와 배우 사이는 일대다 연관 관계다. 영화 한 편은 많은 배우가 있다. 앞에서 일대다 연관 관계를 이미 다루었으므로 여기서는 훑고 지나가겠다.

우선 POJO와 함께 진행해 보자.

```
// Actor
public class Actor {
       private int id = 0;
       private String firstName = null;
       private String lastName = null;
       private String shortName = null;
       // Setters and getters
       …
       }
}
```

단방향 관계를 보고 있으므로 `Actor` POJO는 `Movie` 객체와 어떠한 참조도 갖지 않는다. `Movie` POJO는 다음과 같다.

```
public class Movie {
       private int id = 0;
```

```
        private String title = null;
        private Set<Actor> actors = null;
        public Set<Actor> getActors() {
            return actors;
        }
        public void setActors(Set<Actor> actors) {
            this.actors = actors;
        }
        …
}
```

actors 변수가 중요하다. 영화 한 편에는 여러 명의 배우가 나오므로 Set 타입 변수를 갖도록 Movie 객체를 선언한다.

매핑 정의로 넘어가기 전에 관련 테이블 데이터베이스 스크립트를 살펴보자. Actors 테이블에는 MOVIES 테이블의 기본키를 가리키는 외래키가 있다.

```
// 간단하고 쉬운 movie 정의
CREATE TABLE MOVIE_ONE2MANY (
       MOVIE_ID int(10) NOT NULL,
       TITLE varchar(10) DEFAULT NULL,
       PRIMARY KEY (MOVIE_ID)
)

// 외래키 정의에 주의
CREATE TABLE ACTOR_ONE2MANY (
       ACTOR_ID int(10) NOT NULL AUTO_INCREMENT,
       FIRST_NAME varchar(20) DEFAULT NULL,
       LAST_NAME varchar(20) DEFAULT NULL,
       SHORT_NAME varchar(20) DEFAULT NULL,
       MOVIE_ID int(10) DEFAULT NULL,
       PRIMARY KEY (ACTOR_ID),
```

```
        CONSTRAINT FK_MOV_ID FOREIGN KEY (MOVIE_ID)
        REFERENCESMOVIE_ONE2MANY (MOVIE_ID)
)
```

MOVIE_ONE2MANY 테이블은 간단한 구조지만, ACTOR_ONE2MANY 테이블은 MOVIES 테이블의 기본키를 가리키는 외래키를 정의한다.

마지막으로 매핑 정의다. 테이블 간의 관계 정보를 가지고 있으므로 영화 정보를 통해 배우를 찾을 수도 있고 그 반대도 가능하다. Actor 객체의 정의는 간단하지만, Movie 객체에는 연관 관계가 정의되어 있다.

```
<hibernate-mapping package="com.madhusudhan.jh.associations.one2many">
      <class name="Actor" table="ACTOR_ONE2MANY">
          <id  name="id" column="ACTOR_ID">
              <generator class="assigned"/>
          </id>
          <property name="firstName" column="FIRST_NAME" />
              ...
      </class>
</hibernate-mapping>

// Movie
<hibernate-mapping package="com.madhusudhan.jh.associations.one2many">
      <class name="Movie" table="MOVIE_ONE2MANY">
          <id name="id" column="MOVIE_ID" >
              <generator class="assigned"/>
          </id>
          <property name="title" column="TITLE" />
              <set name="actors" table="ACTOR_ONE2MANY" cascade="all">
              <key column="MOVIE_ID" not-null="true"/>
              <one-to-many class="Actor"/>
          </set>
```

```
        </class>
    </hibernate-mapping>
```

actors 변수와의 연관 관계를 나타내기 위해 set 요소를 정의한다. Movie POJO에 정의한 java.util.Set과 동일하다. 앞의 코드에서는 Movie 객체마다 MOVIE_ID로 질의된 Actor 클래스와 연관 관계인 ACTOR_ONE2MANY 테이블에서 배우 정보를 가져온다.

일대다 매핑 확인을 위해 실행할 테스트 예제를 만들어보자.

```
public class OneToManyTest {
        private Movie persistMovie() {
            Movie movie = null;
            Actor actor = null;
            Set<Actor> actors = new HashSet<Actor>();

            // actors 생성
            actor = new Actor("Sharukh", "Khan", "King Khan");
            actors.add(actor);
            actor = new Actor("Deepika", "Padukone", "Miss Chennai");
            actors.add(actor);

            // Movie 객체 생성하기와 배우 연관시키기
            movie = new Movie("Chennai Express");
            movie.setActors(actors);
            session.save(movie);
            …
        }
}
```

실행 과정이 복잡하지는 않다. Movie 객체의 인스턴스를 생성하고 Actors 객체와 관계를 맺는다. movie 인스턴스를 저장하고, Actors 객체로 구성된 Set 콜렉션은 "cascade=true" 속성을 통해 자동으로 영속화된다.

앞의 예제에는 양방향 관계가 아니므로 배우 정보를 가지고 그 배우에 해당하는 영화 정보를 조회할 수 없다. 양방향 관계를 구현하기 위해서는 POJO와 매핑 정보 변경이 필요하다(테이블은 기존처럼 유지한다).

5.4 양방향 일대다 연관 관계

다음에서 보듯이 Actor 객체에 Movie 객체를 가리키는 참조를 추가해야 한다. 이미 Actors 객체 참조를 가진 Movie POJO는 변경할 필요가 없다.

```
public class Actor {
      // 이 배우가 출연한 영화를 알고 싶다.
      private Movie movie = null;
          …
      public Movie getMovie() {
          return movie;
      }
      public void setMovie(Movie movie) {
          this.movie = movie;
      }
}
```

다음은 Actor.hbm.xml 매핑 파일이다. 이제 many-to-one 요소를 만들어보자.

```
<hibernate-mapping package="com.madhusudhan.jh.associations.one2many.bi">
      <class name="Actor" table="ACTOR_ONE2MANY_BI">
          <id name="id" column="ACTOR_ID">
              <generator class="assigned"/>
          </id>
          <many-to-one name="movie" column="MOVIE_ID" class="Movie"/>
              …
      </class>
</hibernate-mapping>
```

many-to-one 요소는 Actor 클래스의 many를 가리킨다. 이제 Movie 객체를 통해 참여한 배우를 조회할 수 있고, 배우가 참여한 영화를 actor.getMovie 메소드를 호출해서 찾을 수도 있다. 이로써 Movie와 Actors 클래스 사이에 양방향 관계가 성립된다.

5.5 다대다 연관 관계

다대다 연관 관계는 각 관계를 시작하는 클래스가 대상이 되는 클래스의 many(다) 부분을 가지는 방법으로 두 클래스 사이에 관계를 성립시킨다. 반대도 마찬가지다. Student와 Course 예제는 다대다 연관 관계가 성립한다. 학생 한 명이 여러 수업을 등록할 수 있고, 강의 한 개에 많은 학생이 참여할 수 있다. 엄밀히 이야기하면 Movie와 Actor 예제는 다대다 연관 관계다. 영화 한 편에 많은 배우가 참여하고, 배우 한 명은 여러 영화에서 연기하기 때문이다(그러나 단순하게 설명하려고 앞에서는 일대다 관계로 가정했다).

자바에서 다대다를 어떻게 다루는가? 양쪽 클래스 모두 접근 메소드로 콜렉션을 나타내는 속성을 가진다. 하지만 데이터베이스는 다대다 매핑을 위해서 일반적으로 연결 테이블link table이라고 하는 제 3의 테이블을 사용한다.

다음과 같이 POJO 클래스들은 서로를 참조하는 속성을 가진다.

```
// Course 클래스
public class Course {
        private int id = 0;
        private String title = null;
        // 학생들을 가진 콜렉션
        private Set<Student> students = null;
        …
}
```

```
// Student 클래스
public class Student {
	private int id = 0;
	private String name = null;
	// 많은 수업에 등록한 학생
	private Set<Course> courses = null;
	…
}
```

Student와 Course 클래스 간 양방향 다대다 연관 관계를 만들었다. 다음은 이와 관련한 매핑 정보다.

```
<!—Student 매핑 -->
<hibernate-mapping package="com.madhusudhan.jh.associations.many2many">
	<class name="Student" table="STUDENT">
		…
		<set name="courses" table="STUDENT_COURSE" cascade="all">
		<key column="STUDENT_ID" />
		<many-to-many column="COURSE_ID" class="Course"/>
	</set>
	</class>
</hibernate-mapping>

<!-- Course 매핑 -->
<hibernate-mapping package="com.madhusudhan.jh.associations.many2many">
	<class name="Course" table="COURSE">
		…
	<set name="students" table="STUDENT_COURSE" inverse="true" cascade="all">
		<key column="COURSE_ID" />
		<many-to-many column="STUDENT_ID" class="Student"/>
	</set>
	</class>
```

```
</hibernate-mapping>
```

마지막으로 연결 테이블을 생성하면 된다. 스크립트는 다음과 같다.

```
CREATE TABLE student_course (
      COURSE_ID int(10) NOT NULL,
      STUDENT_ID int(10) NOT NULL,
      PRIMARY KEY (COURSE_ID,STUDENT_ID),
      CONSTRAINT fk_course_id FOREIGN KEY (COURSE_ID) REFERENCES course (COURSE_ID),
      CONSTRAINT fk_student_id FOREIGN KEY (STUDENT_ID) REFERENCES student
          (STUDENT_ID)
)
```

5.6 요약

이번 장에서는 객체의 연관 관계를 자세히 살펴보았고, 일대일, 일대다 또는 다대일, 다대다의 차이점을 살펴보았다. 그리고 몇 가지 예제를 통해 다양성과 방향성을 알아보았다.

6 | 고급 개념

앞에서 하이버네이트의 기본 개념을 살펴보았다. 하이버네이트는 기능이 많은 ORM 프레임워크이기 때문에 이를 다 설명하기는 어렵다. 그러나 하이버네이트로 작업할 때 이해해야 할 중요한 기능들이 있는데, 이번 장에서는 이러한 하이버네이트 아주 유용한 특징을 설명하겠다.

6.1 하이버네이트 타입

매핑 설정 파일에서는 `column="COLOR" name="color"` 속성처럼 테이블 컬럼과 객체 속성을 매핑해 준다. 그런데 하이버네이트에서는 `COLOR` 컬럼이 VARCHAR 타입인지 color 속성이 `String` 타입인지 어떻게 알 수 있을까?

속성 타입을 알아내기 위해 자바의 리플렉션을 사용한다. 비록 타입이 생략된 옵션이 잘 동작하더라도 옵션은 속성의 타입을 지정하는 것을 권장한다. `column="COLOR" name="color" type="string"` 속성값을 지정함으로써 하이버네이트에 속성 타입을 쉽게 전달할 수 있다.

color 속성의 타입이 `String`이 아닌 `string`이라는 점에 주목하자. 이 `string` 타입은 자바 타입도 아니고 SQL 타입도 아닌 하이버네이트 자체 타입이다. 하이버네이트는 string, boolean, integer와 같은 내장 타입과 사용자 정의 타입을 포함한 광범위한 타입을 지원한다.

6.1.1 엔티티 타입과 값 타입

하이버네이트 타입 시스템은 기본적으로 엔티티 타입entity type과 값 타입value type으로 분류된다. 가장 주된 차이점은 엔티티 타입은 식별자가 있고 자체로도 존재할 수 있

지만, 값 타입은 그렇지 않다는 것이다. Movie, Car, Showroom, Student 같은 영속화 객체가 엔티티의 예로, 이들은 자신을 고유하게 나타내는 식별자가 있으므로 독립적이다. 하지만 값 타입은 스스로 존재할 수 없고, 엔티티와 같은 객체에 의존적이다. 하이버네이트에서는 기본 타입과 컴포넌트라는 두 종류의 값 타입을 제공한다.

기본 타입

기본 타입Basic Type은 테이블 컬럼과 자바 속성을 매핑하는 데 사용한다. 예를 들어, Movie 객체의 title 속성과 매핑된 string 타입을 보았다. string, boolean, int, long, double, timestamp 등이 기본 타입에 속한다. 기본 타입은 기존의 엔티티(여기서는 Movie)와 연계되어야 한다.

컴포넌트

종종 한 개 이상의 필드로 정의된 타입이 필요할 때가 있다. 컴포넌트Component는 특정 타입으로 구성된 필드의 집합을 정의한다. 예를 들어, 국가 코드, 지역번호, 이름의 집합체로 전화번호를 만들 수 있다. 컬럼/속성으로 구성된 조합을 반복해서 사용해야 할 수도 있다. 그래서 PhoneNumber 컴포넌트를 만들고 이것을 타입으로 서로 연결하는 것이 효율적이다. 컴포넌트는 테이블 정보를 여러 개의 객체로 나눌 때 매우 유용하다.

자바 콜렉션 역시 영속화할 수 있는 타입이다. 자세한 내용은 4장 컬렉션 영속화를 참조하길 바란다.

6.1.2 사용자 정의 타입

이미 살펴본 두 타입 외에도 하이버네이트에서는 사용자 정의 타입을 생성할 수 있다. 앞의 PhoneNumber 타입을 사용해야 한다면 타입을 만들어야 한다. 이를 PhoneNumberType 클래스라 하겠다. 이 클래스는 org.hibernate.type.Type이나 BasicType 같은 하이버네이트의 인터페이스를 구현한다.

org.hibernate.type.BasicType 인터페이스를 구현한 PhoneNumberType 클래스를 만들어보자. 몇 가지 메소드를 더 구현해야 하지만 간략하게 하기 위해 다음과 같이 일부 메소드만 소개한다.

```
public class PhoneNumberType implements BasicType {
     public int[] sqlTypes() {
         return new int[]{
             IntegerType.INSTANCE.sqlType(),
             IntegerType.INSTANCE.sqlType(),
             StringType.INSTANCE.sqlType()
         };
     }
     public Class returnedClass() {
         return PhoneNumber.class;
     }
     …
}
```

sqlTypes() 메소드는 두 개의 integer 타입(국가 코드, 지역번호)과 한 개의 string 타입(이름)으로 구성된 정보를 하이버네이트에 제공한다. 다음으로 새롭게 정의한 타입을 등록하고 그 정보를 하이버네이트에 전달한다. 다음은 Configuration 인스턴스의 registerTypeOverride() 메소드를 호출하는 부분이다.

```
Configuration config = new
Configuration().configure("/types/hibernate.cfg.xml");
config.registerTypeOverride(new PhoneNumberType());
```

한 번 새로운 타입을 정의하면 다음과 같이 매핑 과정에서 그 타입을 클래스의 한 속성으로 사용할 수 있다.

```
<class name="CustomCar" table="CUSTOM_CAR">
    ...
    <property name="phoneNumber" column="PHONE_NUMBER"
    type="com.madhusudhan.jh.advanced.types.PhoneNumberType"/>
    ...
</class>
```

6.2 컴포넌트

테이블에 의존하지 않고 객체 모델에 따라 객체를 구성하고자 할 때가 있다. 100개의 모든 테이블 컬럼에 해당하는 100개의 속성을 가지는 커다란 하나의 객체가 있다면 이는 좋은 설계가 아니다.

다음의 Person 클래스를 살펴보자.

```
public class Person {
    private String firstName = null;
    private String nickName = null;
    private String lastName = null;

    // Phone 상세
    private int areaCode = 0;
    private int phoneNumber = 0;
    private String name = null;
}
```

Person 객체는 전화번호와 관련된 세 개의 속성인 지역번호, 전화번호, 이름을 가지고 있다. 업무상 한 사람이 여러 개의 전화번호를 보유할 수 있다는 새로운 요구사항이 생겼다고 가정해 보자. 이전 설계를 유지한다면 또 다른 속성 세 개를 추가해야 한다(중복된다!). person 객체에 부가적으로 주소 정보를 정의할 때마다 반복적으로 속성들을 가지고 있어야 할까?

다음과 같이 PhoneNumber 클래스를 생성하고 person 객체를 리팩토링할 수 있다.

```
public class PhoneNumber {
     // Phone 상세
     private int areaCode = 0;
     private int phoneNumber = 0;
     private String name = null;
}
```

Person 객체는 다음과 같이 수정된다.

```
public class Person {
     private String firstName = null;
     private String nickName = null;
     private String lastName = null;

     // 복수 개의 Phone 상세
     private PhoneNumber homePhone = null;
     private PhoneNumber mobilePhone = null;
}
```

간단한 리팩토링이지 않은가? 자바 엔티티를 리팩토링하는 것이 놀라운 일은 아니다. 테이블 구조를 전혀 수정하지 않았다(Person 테이블에 있던 전화번호 관련 필드 세 개가 여전히 존재한다).

그런데 새롭게 만들어진 PhoneNumber 클래스와 개선된 person 객체 간의 매핑 정보는 어떻게 만들어야 할까? 이제 컴포넌트가 등장할 때다. 컴포넌트는 그룹 컬럼을 객체로 넣는 역할을 한다. 앞의 예제에서 PhoneNumber는 엔티티 클래스가 아니라 단순히 테이블 컬럼을 나타내는 클래스였다. Person 클래스와 PhoneNumber 클래스 간 관계는 정확히 일대일 부모-자식 관계다.

매핑 정보는 다음과 같다.

```
<class name="Person" table="PERSON">
     <id name="id" column="PERSON_ID">
        <generator class="native"/>
     </id>
     <property name="firstName" column="FIRST_NAME" />
     …
     <component name="homePhone" class="PhoneNumber">
        <property name="areaCode" column="HOME_AREA_CODE"/>
        <property name="phoneNumber" column="HOME_PHONE_NUMBER"/>
        <property name="name" column="HOME_NAME"/>
     </component>

     <component name="mobilePhone" class="PhoneNumber">
        <property name="areaCode" column="MOBILE_AREA_CODE"/>
        <property name="phoneNumber" column="MOBILE_PHONE_NUMBER"/>
        <property name="name" column="MOBILE_NAME"/>
     </component>
</class>
```

매핑 정보에서 phoneNumber 변수를 나타내기 위해 component 태그를 사용한다. 실제 PhoneNumber 클래스를 참조하는 클래스 속성과 특정 컬럼을 참조하는 PhoneNumber 속성에 주목해 보자. Person 코드에서 마치 한 개의 컬럼처럼 사용되는 phoneNumber 변수를 볼 수 있다. 집 전화와 모바일 전화를 위한 두 개의 컴포넌트를 정의한다.

person.getPhoneNumber().getName() 메소드 같은 표준 형식으로 간단히 컴포넌트 변수에 접근한다. 컴포넌트는 필드를 잘게 나뉜 도메인 모델로 그룹화하는 데 매우 유용하다.

6.3 캐싱

성능 최적화를 한다면 캐싱 방법이 가장 먼저 떠오른다. 캐싱으로 데이터 중심 애플리케이션의 성능을 개선할 수 있다. 하이버네이트에서는 1차와 2차 캐싱 메소드를 사용하여 영속화 객체를 캐싱하도록 지원한다.

6.3.1 1차 캐시

1차 캐시는 Session 객체와 관련하여 트랜잭션이 보장되는 캐시다. 이는 세션의 수명이 유지되는 동안이나 컨버세이션conversation 내에서만 가능하다. 1차 캐시는 하이버네이트 프레임워크에서 기본으로 제공한다. 다음 코드를 살펴보자.

```
private void firstLevelCache() {
      …
      int personId = 10;
      Person person = new Person();
      person.setId(personId);
      person.setFirstName("Madhusudhan");
      person.setLastName("Konda");
      session.save(person);

      // 닉네임을 지정하기 위해 같은 객체를 불러보자
      // 동일한 세션을 이용한다는 점을 기억하자
      person = (Person) session.load(Person.class, personId);
      person.setNickName("MK2");
      session.save(person);
}
```

앞의 코드에서 우선 Person 인스턴스를 생성하고, 저장하기 전에 값들을 지정한다. 동일한 Session 인스턴스를 가지고 person 객체를 가져와서 다시 또 다른 속성(여기서는 nickName)을 지정한다. 두 번째로 person 객체를 불렀을 때 세션 자체

가 가진 캐시에서 해당 객체를 조회한다. 이렇게 데이터베이스를 이용하는 네트워크 라운드트립roundtrip을 피한다. 세션 캐시는 클래스 타입과 함께 명시되므로 이미 존재하는 인스턴스를 오버라이드할 때 좀 더 주의해야 한다.

6.3.2 2차 캐시

2차 캐시는 SessionFactory 클래스를 이용하여 전역에서 사용할 수 있다. 그래서 2차 캐시 안에 있는 어떤 데이터라도 애플리케이션 전체에서 사용이 가능하다.

하이버네이트는 EhCache와 InfiniSpan 같은 오픈소스 캐시 라이브러리를 지원한다. 사용자 정의 캐시 라이브러리를 사용하려면 org.hibernate.cache.spi.CacheProvider 인터페이스 관련 라이브러리를 구현하면 된다. 하이버네이트는 기본 옵션으로 EhCache를 2차 캐시 공급자로 사용한다.

캐시 공급자를 연결하려면 hibernate.cache.provider 클래스 속성에 캐시 공급자를 명시한다. 다음은 JBoss의 InifiniSpan을 캐시 공급자로 연결하는 예다.

```
<hibernate-configuration>
    <session-factory>
        <!-- Infinispan 캐시 공급자 지정 -->
        <property name="hibernate.cache.provider_class">
        org.hibernate.cache.infinispan.InfinispanRegionFactory
        </property>
    …
    </session-factory>
</hibernate-configuration>
```

다양한 캐시 속성을 이용하여 캐싱할 클래스별로 캐싱 정책을 지정할 수 있다. 매핑 정의 파일의 class 태그 안에 cache 속성을 확인해 보자.

```
<hibernate-mapping package="com.madhusudhan.jh.advanced.cache">
    <class name="Person" table="PERSON">
        <cache usage="read-write" region="" include="all"/>
        ...
    </class>
</hibernate-mapping>
```

usage는 필수 요소이므로 캐시 동시성 방법을 지정해야 한다(앞의 예제는 read-write를 사용한다). usage 속성에는 네 가지 설정 방법이 있다.

transactional

이 전략은 트랜잭션이 가능한 캐시를 지원하는 캐시 공급자를 위해 제공된다. 모든 캐시 공급자가 트랜잭션이 가능한 캐싱 상품을 가지고 있지 않다는 점에 유의하자.

read-only

갱신할 필요가 없는 영속화 객체에 자주 접근한다면 read-only를 선택한다. 데이터베이스를 거의 변경하지 않거나 전혀 변경하지 않는다면 이 옵션으로 성능이 크게 향상할 것이다.

read-write

객체를 데이터베이스에서 읽거나 데이터베이스에 쓸 때 이 방법을 사용한다.

nonstrict-read-write

객체를 그다지 자주 갱신하지 않을 때 사용한다.

이 옵션을 전역에서 사용하려면 설정 파일에서 hibernate.cache.default_cache_concurrency_strategy 속성을 설정한다.

6.3.3 쿼리 캐시

객체 뿐만 아니라 쿼리도 캐싱할 수도 있다. 특정 쿼리를 빈번하게 사용한다면 캐싱하는 것을 추천한다. 이 기능을 사용하려면 `hibernate.cache.use_query_cache` 속성을 `true`로 지정한다. 코드에 한 가지를 더 추가해야 하는데, `Query.setCacheable()` 메소드를 호출해서 `Query`의 `cacheable` 속성을 `true`로 지정해야 한다.

6.4 상속 전략

객체지향 프로그래밍을 생각할 때, 바로 떠오르는 원칙은 상속일 것이다. 현실 문제들을 모델링할 때 언제나 has-a나 is-a를 고민할 것이다. 예를 들어, `Executive`는 `Employee`면서(is-a) `Address`를 가지고 있다(has-a). has-a 상속 관계를 지원하기 위해 기본키와 외래키를 이용하더라도 관계형 데이터베이스에서는 is-a 상속 관계를 알 길이 없다. 이는 ORM 툴에서 상속 관계를 영속화하려고 할 때 약간의 문제를 발생시킨다. 하이버네이트에서는 상속 구조의 영속성을 지원하기 위해서 세 가지 전략을 제공함으로써 이 문제를 해결했다. 각 전략을 살펴보자.

6.4.1 Table-per-Class 전략

Table-per-class 전략에서는 모든 객체 계층을 위해 한 개의 테이블을 정의한다. 이는 간단한 전략으로, 단일 테이블로도 애플리케이션의 데이터를 저장하는 데 충분하다. 예를 들어, `Employee`와 `Executive` 객체 데이터는 같은 테이블에 영속화된다.

그런데 어떻게 데이터를 분리할 수 있을까? 어떻게 사원의 정보와 경영진의 정보를 구별할 수 있을까? 구별자discriminator 컬럼을 도입해서 구별할 수 있다. 구별자 컬럼은 클래스마다 개별적으로 데이터를 표시한다.

[표 6-1] Table-per-class 전략

ID	NAME	ROLE	Discriminator
5	Barry Bumbles	NULL	EMPLOYEE
6	Harry Dumbles	Operations	EXECUTIVE

테이블 안에 두 개의 로우가 존재하지만, 구별자 컬럼은 해당 컬럼의 값으로 각 로우를 구별한다. 첫 번째 로우는 Employee의 정보고, 두 번째 로우는 Executive의 정보라는 것을 알려준다.

하이버네이트에서 이 전략을 구현하는 방법을 살펴보자. Employee와 Executive에 해당하는 두 클래스가 있고 Executive 클래스는 Employee 클래스를 상속받는다.

```
// 부모 클래스
public class Employee {
     private int id = 0;
     private String name = null;
     …
}

// 자식 클래스
public class Executive extends Employee{
     private String role = null;
     …
}
```

XML 매핑을 이용한 Table-per-class 전략

영속화 클래스가 있으므로 다음 단계는 매핑 정보를 정의한다. 다음 매핑 정의 파일 EmployeeExecutive.hbm.xml을 살펴보자.

```
<hibernate-mapping package="com.madhusudhan.jh.advanced.inheritance.s1">
     <class name="Employee" table="INHERITANCE_S1_EMPLOYEE" discriminator-
```

```
value="EMPLOYEE">
        <id name="id" column="EMPLOYEE_ID">
            <generator class="native"/>
        </id>
        <discriminator column="DISCRIMINATOR" type="string"/>
        <property name="name" column="NAME" />
        <subclass name="Executive" extends="Employee" discriminator-
value="EXECUTIVE">
            <property name="role" column="ROLE"/>
        </subclass>
    </class>
</hibernate-mapping>
```

테이블을 가리키는 Employee 클래스를 선언한다. 이 class 요소에 정의된 discriminator-value 태그에 주목해 보자. 이 값은 경영진 객체와 함께 DISCRIMINATOR 컬럼에 영속화되는 정적static 값이다. 해당 컬럼은 discriminator라는 또 다른 속성을 통해서 표현된다. 그리고 discriminator는 id 요소 이후에서만 선언된다.

Executive 클래스의 매핑 정의를 살펴보자. 정의 부분에서 extends="Employee"는 클래스 간의 계층을 가리키기 위해 subclass 요소를 사용한다. Employee에서처럼 "EXECUTIVE" 정적 값을 지정하고, discriminator-value 속성에 정의된 값으로 경영진 객체를 영속화한다.

다음 테이블 정의를 살펴보자.

```
create table inheritance_s1_employee(
    employee_id         int not null auto_increment,
    name                varchar(20) not null,
    role                varchar(20),
```

```
discriminator        varchar(20),
primary              key (employee_id));
```

즉, 테이블의 구별자 컬럼은 매핑 설정 파일에 정의된 EXECUTIVE와 EMPLOYEE 두 값을 저장하기 위한 컬럼이다.

이번에는 어노테이션을 이용한 방법을 살펴보자.

어노테이션을 이용한 Table-per-class 전략

어노테이션을 이용한 Table-per-class 상속 전략은 복잡하지 않다. 어노테이션으로 부모 클래스 Employee를 생성하자.

```
@Entity(name="INHERITANCE_S1_EMPLOYEE_ANN")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISCRIMINATOR" discriminatoryType=DicriminatorType
.STRING)
@DiscriminatorValue(value="EMPLOYEE")

public class Employee {
     @Id
     @Column(name="EMPLOYEE_ID")
     private int id = 0;
     …
}
```

@Inheritance 어노테이션을 엔티티에 명시함으로써 Table-per-class 상속 전략을 정의한다. 이 어노테이션의 경우 strategy 변수를 통해 전략을 지정하는데, 여기서는 SINGLE_TABLE 전략을 사용한다. 그 외에도 InheritanceType에는 TABLE_PER_CLASS와 JOINED 전략이 있다. Table-per-class 전략을 이용할 때 InheritanceType.TABLE_PER_CLASS 값으로 잘못 지정할 수도 있다. 반드시

SINGLE_TABLE만 사용하자. TABLE_PER_CLASS 값은 Table-per-concrete-class 전략에서 사용하며 뒤에서 살펴볼 것이다.

상속 전략과 함께 @DiscriminatorColumn 어노테이션으로 구별자 컬럼도 정의해야 한다. 이 어노테이션에서는 구별자 컬럼의 이름과 타입을 만들어 준다. @DiscriminatorValue 어노테이션은 엔티티에 정적 값을 지정하는데, 이 경우에는 EMPLOYEE 값이다.

자식 클래스 Executive의 어노테이션은 간단하다. @DiscriminatorValue 값은 영속화되는 모든 경영진 객체에서는 정적 값이 EXECUTIVE로 지정된다.

```
@Entity
@DiscriminatorValue(value="EXECUTIVE")
public class Executive extends Employee {
     private String role = null;
     …
}
```

이제 마지막으로 설정 파일을 통해 어노테이션된 클래스를 추가해 보자.

```
public class InheritanceStrategyOneTest {
     private void init() {
     Configuration config = new Configuration()
         .configure("advanced/inheritance/s1/hibernate.cfg.ann.xml")
         .addAnnotatedClass(Employee.class)
         .addAnnotatedClass(Executive.class);
         …
     }

     private void test() {
         Employee emp = new Employee("Barry Bumbles");
```

```
            session.save(emp);
            Executive ex = new Executive("Harry Dumbles");
            ex.setRole("Director");
            session.save(ex);
            …
        }
    }
```

테스트 데이터를 영속화하기 위한 Table-per-class 전략을 보여준다. 이 전략은 간단한 상속 계층 구조에 알맞은 전략이다. 많은 단계의 객체 그래프object graph를 갖기 시작하면 이 전략은 해당 관계와 맞지 않게 되고 유지하는 데 적합하지 않게 된다. 도메인 클래스가 변경되면 테이블 구조 역시 수정되어야 한다. Table-per-class 전략에서는 하위 클래스와 관련된 컬럼에는 NOT NULL 제약사항을 선언하지 못한다는 점에 주의하자.

6.4.2 Table-per-Subclass 전략

앞에서 단일 테이블에 모든 행을 영속화하는 Table-per-class 전략을 살펴보았는데, 구별자 컬럼을 이용하여 각 로우를 구분할 수 있었다. 객체 그래프object graphs를 저장하는 하나의 큰 테이블을 이용하는 대신에 분리된 테이블을 사용하는 방법도 있다. 이를 Table-per-subclass 상속 영속화 전략이라 한다.

이 전략에서는 모든 하위 클래스(추상 클래스를 제외한 부모 클래스도 포함된다)가 영속화 테이블을 가지고 있다. 사원과 경영진 객체는 각각 EMPLOYEE와 EXECUTIVE 테이블에 영속화되는데, 이 방법에서는 구별자 컬럼을 정의하지 않는다. 그러나 하위 클래스의 테이블은 부모 클래스의 기본키를 참조하는 외래키를 가지고 있다는 점에 주의하자.

Table-per-subclass 전략을 보여주는 예제를 살펴보자.

```
// EMPLOYEE 테이블
CREATE TABLE inheritance_s2_employee (
     EMPLOYEE_ID int(11) NOT NULL AUTO_INCREMENT,
     NAME varchar(255) DEFAULT NULL,
     PRIMARY KEY (EMPLOYEE_ID)
)

// EXECUTIVE 테이블
CREATE TABLE inheritance_s2_executive (
     EMPLOYEE_ID int(11) NOT NULL,
     ROLE varchar(255) DEFAULT NULL,
     PRIMARY KEY (EMPLOYEE_ID),
     CONSTRAINT FK_EMP FOREIGN KEY (EMPLOYEE_ID)
     REFERENCES inheritance_s2_employee (EMPLOYEE_ID)
)
```

EMPLOYEE 테이블은 기준이 되는 테이블이다. 하지만 하위 클래스인 Executive에 해당하는 EXECUTIVE 테이블에는 제약사항이 정의되어 있다.

XML 매핑 정의를 이용한 Table-per-subclass 전략

동일한 계층 구조이므로 Employee와 Executive 클래스 정의에는 변경사항이 없지만, 매핑 정의는 수정해야 한다.

```
<hibernate-mapping package="com.madhusudhan.jh.advanced.inheritance.s2">
     <class name="Employee" table="INHERITANCE_S2_EMPLOYEE">
        <id name="id" column="EMPLOYEE_ID">
           <generator class="native"/>
         </id>
        <property name="name" column="NAME" />
        <joined-subclass name="Executive" table="INHERITANCE_S2_EXECUTIVE">
           <key column="EMPLOYEE_ID"/>
```

```
            <property name="role" column="ROLE"/>
        </joined-subclass>
    </class>
</hibernate-mapping>
```

부모 클래스 Employee는 설명한대로 정의되었지만, 자식 클래스 Executive는 joined-class 요소와 함께 명시되었다. 이제 이 하위 클래스는 INHERITANCE_S2_EXECUTIVE 테이블을 가진다. joined-subclass의 key 속성을 이용하여 어떻게 외래키와 연결되는지에 주목하자.

변경사항은 이게 전부다. 테스트 프로그램을 실행하면 다음과 같은 결과가 나온다.

```
// EMPLOYEE 테이블
ID   NAME
1    Barry Bumbles
2    Harry Dumbles

// EXECUTIVE 테이블
ID   NAME
2    Director
```

데이터를 구분하는 구별자 컬럼을 가진 이전 예제의 단일 테이블과 달리 클래스마다 한 개씩, 총 두 개의 테이블이 존재한다. EXECUTIVE 테이블은 부모 클래스의 기본키인 EMPLOYEE 테이블의 id를 바라보는 외래키를 가지고 있다.

어노테이션을 이용한 Table-per-subclass

어노테이션을 이용해도 동일한 작업 결과를 얻을 수 있다. 부모 클래스에 다음의 어노테이션을 선언해 보자.

```
Entity(name="INHERITANCE_S2_EMPLOYEE_ANN")
```

```
@Inheritance(strategy= InheritanceType.JOINED)
public class Employee {
     Id
     @Column(name="EMPLOYEE_ID")
     private int id = 0;
     ...
}
```

InheritanceType을 JOINED 값으로 지정해서 Table-per-subclass 전략을 설정한다. 자식 클래스에는 @PrimaryKeyJoinColumn 어노테이션을 이용하여 다음과 같이 외래키인 주primary 조인 컬럼을 선언해야 한다.

```
@Entity(name="INHERITANCE_S2_EXECUTIVE_ANN")
@PrimaryKeyJoinColumn(name="EMPLOYEE_ID")
public class Executive extends Employee
{
     ...
}
```

테스트 프로그램을 실행하면 두 테이블에는 사원과 경영진 정보가 들어간다.

다음으로 하이버네이트에서 제공하는 상속의 세 번째 영속화 전략을 살펴보자.

6.4.3 Table-per-Concrete-Class 전략

Table-per-concrete-class 전략에서 객체의 계층 구조는 각 구현 클래스에 대응하는 개별 테이블에 영속화된다. 상위 클래스의 모든 속성이 자식 클래스와 관련된 테이블에 복사된다. 그렇기 때문에 이 전략은 일반적이지는 않다. 실제로 실행하면서 이 전략을 살펴보자.

새로운 상위 클래스인 Person 추상 클래스에 맞게 객체 그래프를 약간 변경해야

한다. Employee와 Executive 클래스는 Person 클래스를 상속받지만, 서로는 상속하지 않는다.

계층 구조는 다음과 같이 정의된다.

```
public abstract class Person {
     private int id;
     private String name = null;
     …
}
public class Employee extends Person{
     private String role = null;
     …
}
public class Executive extends Person{
     private double bonus = 0.0;
     …
}
```

XML 매핑을 이용한 Table-per-concrete-class 전략

매핑 정의 파일과 관련된 변경사항은 다음과 같다.

```
<hibernate-mapping package="com.madhusudhan.jh.advanced.inheritance.s3">
     <class name="Person" abstract="true">
        <id name="id" column="EMPLOYEE_ID">
           <generator class="assigned"/>
        </id>
        <property name="name" column="NAME" />
        <union-subclass name="Employee" table="INHERITANCE_S3_EMPLOYEE">
           <property name="role" column="ROLE"/>
        </union-subclass>
```

```
        <union-subclass name="Executive" table="INHERITANCE_S3_EXECUTIVE">
            <property name="bonus" column="BONUS"/>
        </union-subclass>
    </class>
</hibernate-mapping>
```

추상 클래스인 Person 클래스는 abstract="true" 속성을 가진다. Person 클래스를 통해 정의된 id는 자식 클래스에 공유된다. 따라서 Employee와 Executive 클래스의 매핑 정의에 기본키는 별도로 명시하지 않는다.

여기서 union-class 요소를 이용하여서 부모 클래스와 하위 클래스를 연결한다. 이 요소에 자식 클래스와 연결된 테이블을 선언한다. 자식 클래스와 관련된 속성은 union-subclass 요소 내에 정의해야 한다. 그리고 class 요소 레벨에서 선언된 모든 속성은 모든 하위 클래스에 공유된다. native 식별자 전략의 사용은 Table-per-concrete-class 전략에서는 허용되지 않는다.

테스트 프로그램을 실행하면 두 테이블에 데이터가 들어간다. 각 하위 클래스 자신만의 속성 외에도 부모 클래스에서 정의된 name 속성값도 두 테이블에 중복된다. 이는 Table-per-concrete-class 전략의 단점이다.

어노테이션을 이용한 Table-per-concrete-class 전략

마지막으로, 어노테이션을 이용한 Table-per-concrete 상속 전략을 생성하는 방법을 살펴보자. Person 클래스가 Person 클래스를 위한 영속화 테이블을 가지고 있지는 않지만, @Entity 어노테이션을 명시해 주고, TABLE_PER_CLASS로 상속 전략을 지정해 준다. Person 엔티티를 보면 테스트 프로그램에서 사용할 수 있는 위치에 어노테이션을 이용한 클래스들을 놓고 테스트 프로그램을 실행한다(이전 예제를 참고하라).

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Person {
     @Id
     @GeneratedValue
     @Column(name="EMPLOYEE_ID")
     private int id;
     …
}

// Employee 클래스
@Entity(name="INHERITANCE_S3_EMPLOYEE_ANN")
public class Employee extends Person{
     private String role = null;

// Executive 클래스
@Entity(name = "INHERITANCE_S3_EXECUTIVE_ANN")
public class Executive extends Person {
     private double bonus = 0.0;
     …
 }
```

6.5 필터

때로는 애플리케이션에서 데이터 집합 모두가 필요하지 않을 때가 있다. 즉, 부분 집합으로도 충분할 수 있다. 제조사(토요타, BMW 등), 자동차의 색깔(빨간색, 흰색 등), 연식에 따른 자동차 모델을 찾고 싶은 고객에게 모든 정보를 제공하고는 필요 없는 정보를 고객이 알아서 거르게 한다면 이는 자원의 낭비다.

하이버네이트는 SQL의 where 절과 유사하지만 좀 더 동적인 수행을 하는 필터링 기능을 제공한다. 이 기능을 어떻게 사용하는지 살펴보자.

필터 생성은 필터 정의 생성과 코드 내 필터 활성화 두 단계가 있다.

6.5.1 필터 정의 생성하기

매핑 정의 파일에서 filter-def 요소를 이용하여 필터를 정의하고, 필터를 적용할 각 클래스가 서로 연결된다. 요구사항에 따라 다양한 필터 정의를 할 수 있다. 필터 정의는 필터 쿼리에서 매개변수가 될 각 속성의 타입을 정의하는 것으로 구성된다.

다음은 필터를 정의하는 예다.

```
<hibernate-mapping package="com.madhusudhan.jh.advanced.filters">
    <!-- 제조사(Make)로 필터 -->
    <filter-def name="filterByMake">
       <filter-param  name="make" type="string"/>
    </filter-def>

 <!-- 색상(color)으로 필터 -->
    <filter-def name="filterByColour">
       <filter-param  name="color" type="string"/>
    </filter-def>

     <!-- 연식(Age)로 필터 -->
    <filter-def name="filterByAge">
       <filter-param  name="age" type="integer"/>
    </filter-def>

     <!--  제조사(Make)와 모델(Model)로 필터 -->
    <filter-def name="filterByMakeAndModel">
       <filter-param  name="make" type="string"/>
        <filter-param  name="model" type="string"/>
    </filter-def>
    ...
```

```
</hibernate-mapping>
```

필터 네 개를 정의한다. filter-param은 질의할 컬럼의 이름을 정의한다. 그러므로 filterByMake 필터에서는 컬럼명 make에 맞는 매개변수 값을 프로그램에서 지정할 것이라고 예상할 수 있다. 쿼리문은 select * from CARS where make= ?로 변환될 것이다. ?는 검색할 바인딩 변수를 지정하는 기호다.

원하는 만큼 매핑 정의 파일에 필터를 생성할 수 있다. 필터를 정의하면 반드시 엔티티 매핑 정의에 필터 매핑 정의를 연결시킨다.

```
<hibernate-mapping package="com.madhusudhan.jh.advanced.filters">
    <filter-def name="filterByMake">
        ...
    </filter-def>
    <class name="Car" table="FILTERS_CAR">
        <id column="CAR_ID" name="id">
            <generator class="native"/>
        </id>
        <property column="COLOR" name="color"/>
        <property column="NAME" name="name"/>
        <property column="MAKE" name="make"/>
        <property column="MODEL" name="model"/>
            <filter name="filterByMake" condition="make = :make"/>
    </class>
</hibernate-mapping>
```

class 요소 정의에서 filter 요소를 이용하여 필터를 지정한다. 이때 두 가지 속성, name과 condition이 필요하다. name 속성은 filter-def 선언에서 정의한 이름과 맞춘다. conditon 속성은 쿼리에서 지정한 네임드named 매개변수와 동일한 형태를 따른다. 실제 where 절을 정의하는 위치다. :make는 고객이 지정해야 할 매

개변수다. 모든 필터를 사용할 필요는 없는데, 이 경우에는 오직 한 가지 필터만 사용한다.

6.5.2 필터 활성화하기

모든 내용을 다 넣고 필터의 사용 방법을 살펴보자.

```
private void test() {
     Filter filter = session.enableFilter("filterByMake");
     filter.setParameter("make", "BMW");
     List results = session.createQuery("from Car").list();
}
```

먼저 session 인스턴스에서 enableFilter() 메소드를 호출하여 필터를 활성화해야 한다. 이 메소드에서는 인자값으로 이미 정의된 필터명(여기서는 filterByMake)을 지정한다. 필터가 활성화되면 검색 매개변수를 지정하고, 질의문과 그 결과를 생성하기 위해 session 인스턴스를 이용하여 진행한다.

6.6 관계 소유자

일대다 또는 다대다 관계에서는 둘 중 어느 한 쪽에서 관계를 관리하는 책임을 맡아야 한다. Movie와 Actor 예제에서 Movie 매핑은 다음과 같다.

```
<class name="Movie" table="MOVIE_ONE2MANY">
     …
     <set name="actors" table="ACTOR_ONE2MANY" inverse="false" cascade="all">
        <key column="MOVIE_ID" not-null="true"/>
            <one-to-many class="Actor"/>
     </set>
</class>
```

기본 옵션은 inverse="false"다. 그러므로 inverse 관계를 false로 지정하려면 inverse 속성을 명시해야 한다.

이 예제에서 inverse 속성을 false로 하여 ACTORS 테이블이 아닌 MOVIES 테이블이 테이블 간 관계를 관리한다고 의미를 부여했다. 이는 Movie 테이블이 ACTORS 테이블의 외래키 MOVIE_ID를 갱신하는 역할을 맡고 있다는 의미다.

이를 충족하기 위해 하이버네이트는 세 개의 SQL문(두 개의 INSERT 문과 한 개의 UPDATE 문)을 만들어낸다. 두 개의 INSERT 문은 MOVIES와 ACTORS 테이블로 데이터를 넣는 질의문이다. 그리고 UPDATE 문은 ACTORS 테이블에 관계 정보를 갱신한다(ACTORS 테이블에 movie_id 값을 지정한다). 세 번째 SQL문은 중복되거나 아예 사용하지 않을 수도 있다(inverse 속성을 true로 설정하면 사용하지 않을 수 있다는 뜻이다).

inverse 속성을 true로 지정하면 반대로 동작하여 관계 소유자가 ACTORS 테이블에 있다는 뜻이다. 그래서 ACTORS 테이블에서 테이블 간의 관계를 관리한다. 이 경우 두 INSERT 문만 발생하고, 중복되는 세 번째 UPDATE 문은 제거되어 성능이 좋아진다.

6.7 엔티티 연쇄 적용

객체 그래프를 영속화할 때 각 엔티티에 save(또는 update) 명령문을 사용한다. 그러나 객체 그래프에 명시된 cascade 속성은 객체 그래프를 따로따로 저장해야 하는 걱정 없이 모든 객체 그래프를 저장한다. 다음은 엔티티별로 영속화하는 예제다.

```
public void nonCascadingSave(){
      …
      session.saveOrUpdate(movie);
      session.saveOrUpdate(actors);
}
```

코드를 줄이고 좀 더 간단명료하게 하기 위해서 연쇄cascading 적용을 설정하는 것이 효과적이다. 부모 노드만 저장하면 되며, 그래프의 남은 부분은 하이네이트 실행 시 다뤄진다. 다음과 같이 콜렉션 속성에 cascade 속성을 지정한다.

```
<class name="Movie" table="MOVIE_ONE2MANY">
    ...
    <set name="actors" table="ACTOR_ONE2MANY" cascade="save-update"
inverse="false" >
        <key column="MOVIE_ID" not-null="true"/>
            <one-to-many class="Actor"/>
    </set>
</class>
```

따라서 해당 메소드는 한 줄의 save 문으로 줄어든다.

```
public void cascadingSave(){
    ...
    // Movie는 모든 actor를 가지고 있다
    session.saveOrUpdate(movie);
}
```

이 경우에 session.saveOrUpdate(actors)는 중복이다. Movie 부모 객체가 저장되면 actors도 자동으로 저장된다.

객체 그래프 삭제에서도 마찬가지다.

```
public void cascadingDelete(){
    ...
    // 영화(movie)와 관련 배우(actors)를 모두 지운다
    session.delete(movie);
```

```
}
// XML 매핑 정의
<class name="Movie" table="MOVIE_ONE2MANY">
     …
     <set name="actors" table="ACTOR_ONE2MANY" cascade="delete"
inverse="false" >
        <key column="MOVIE_ID" not-null="true"/>
           <one-to-many class="Actor"/>
     </set>
</class>
```

Cascade 속성을 delete로 지정하면 movie 그래프를 완전히 삭제된다. 또한, movie와 movie와 연관된 actor도 마찬가지다.

한 가지가 더 있다. 고아 객체 제거delete-orphan라는 특별한 경우다. 영화/배우 예제에서 set 콜렉션의 actor 객체를 제거하면 movie와 actor 관계가 분리될 뿐만 아니라 actor 엔티티도 완전히 삭제할 것이다. 이 옵션을 선택하지 않으면 movie와 actor 간의 관계는 없어지지만, actor 엔티티는 남는다. 이런 이유로 고아 객체라고 한다.

```
// XML 매핑 정의
<class name="Movie" table="MOVIE_ONE2MANY">
     …
     <set name="actors" table="ACTOR_ONE2MANY" cascade="delete-orphan" >
        <key column="MOVIE_ID" not-null="true"/>
        <one-to-many class="Actor"/>
     </set>
</class>
```

6.8 요약

이 장에서는 상속과 캐싱, 필터, 타입 시스템부터 inverse와 연쇄 기능과 같은 중요한 개념까지 여러 고급 개념을 살펴보았다. 이를 이해했다면 훌륭한 하이버네이트 개발자가 될 것이다.

7 | 하이버네이트 질의어

하이버네이트는 자체 질의어인 하이버네이트 질의어HQL, Hibernate Query Language를 도입했다. HQL은 자바 객체 모델을 질의하기 위해 특별히 설계되었으며, SQL과 비슷한 쿼리어다. 객체 관련 작업을 할 때 HQL을 사용하는데, 이 장에서는 HQL에서 제공하는 기능을 자세히 살펴보겠다.

보통 관계형 테이블을 질의할 때는 SQL을 사용하는데, SQL은 관계형 데이터베이스 작업의 표준이다. 그런데 하이버네이트는 객체 모델 작업을 매끄럽게 진행하기 위해 단순하고 사용하기 쉬운 언어로 만들어졌다. 이런 이유로 SQL을 따라 자바 객체 간 관계 정보를 질의하기 위한 HQL을 만들었다. HQL은 간단하지만 강력한 기능을 제공하는 툴킷이다.

HQL은 SQL과 비슷해서 쉽게 HQL을 배울 수 있다. 예를 들어, `MOVIES` 테이블의 모든 레코드를 조회할 때, SQL에서는 `"SELECT * FROM MOVIES"`를 사용하고 HQL에서는 `"FROM Movie"`를 사용한다. HQL은 객체를 다루므로 테이블을 나타내는 엔티티 객체 클래스명을 사용해야 한다. 이때 `Movie`는 `MOVIES` 테이블에 연결된 영속성 자바 엔티티다. `"FROM com.madhusudhan.jh.hql.Movie"`와 같이 `Movie` 클래스의 전체 패키지명FQN, fully qualified name을 사용할 수 있다. HQL에서 테이블의 모든 데이터 컬럼을 질의할 때 `SELECT` 문을 선택적으로 사용하지만, 한 개 이상의 컬럼을 개별적으로 사용할 때는 반드시 명시해야 한다. 이것을 프로젝션projection이라 한다.

HQL에서는 `WHERE`, `ORDER BY`, `AVG`, `MAX` 등을 SQL처럼 사용할 수 있다. 이 장에서는 이 내용을 다룬다. 우선 기본적인 쿼리 API를 살펴보자.

7.1 Query 클래스 사용하기

하이버네이트에서는 객체 관계 쿼리를 사용하기 위해 Query API를 제공한다. Query 클래스는 간단한 인터페이스로 되어 있고, Query API에서 중요한 부분이다.

Query API로 무언가를 하려면 먼저 Query 클래스로부터 인스턴스를 얻어야 한다. 현재 session 객체에서 createQuery() 메소드를 호출해서 Query 인스턴스를 얻을 수 있다. Query 인스턴스를 얻으면 인스턴스의 메소드를 이용하여 데이터베이스에서 만들어진 결과에 접근할 수 있다.

다음은 어떻게 Query 인스턴스를 얻는지 보여준다.

```
// 현재 session 가져오기
Session session = sessionFactory.getSesion();
// 현재 session에서 query 인스턴스 만들기
Query query = session.createQuery("from TravelReview");
```

세션 팩토리에서 세션 객체를 가져온다. 다음으로 데이터베이스에서 실행할 쿼리를 인자로 전달하고 session.createQuery() 메소드를 호출한다. createQuery() 메소드는 예제에서 보는 것처럼 문자열로 표현된 쿼리문을 받는다.

나머지는 어떻게 진행되는지 살펴보자. "from TravelReview" 문자열은 쿼리를 표현하는 인자다. 이는 TravelReview에 연결된 테이블에서 모든 travelReview 정보를 조회하라는 의미와 같다. 그래서 하이버네이트에서는 HQL 문자열 쿼리를 SQL 쿼리로 변환한다(데이터베이스에서는 오직 SQL 쿼리만 이해할 수 있다). 그리고 SQL을 실행하여 TRAVEL_REVIEW 테이블에서 모든 행을 가져온다. 이는 SQL의 "SELECT * from TRAVEL_REVIEW"와 유사하다.

그러나 몇 가지 작은 차이점이 있다.

- HQL에서는 테이블의 모든 열을 조회할 때 select 키워드는 무시(선택적)되지만, SQL은 그렇지 않다.
- SQL 구문에서는 관계형 테이블(TRAVEL_REVIEW)을 명시하지만, HQL에서는 클래스명을 사용한다. TravelReview는 TRAVEL_REVIEW를 나타내는 영속성 클래스다.

지금까지 세션에서 Query 인스턴스를 얻었다. 이제는 Query 인스턴스에서 데이터를 가져오겠다. 데이터를 다루기 위해 Query API를 사용한다. query API의 list() 메소드를 이용하여 모든 레코드를 가져오겠다.

7.1.1 모든 로우 가져오기

테이블에서 모든 로우를 가져오는 것은 매우 간단하다. 다음 메소드는 TRAVEL_REVIEW 테이블에서 모든 레코드를 가져온다.

```
private void getAllTravelReviews() {
     Query query = session.createQuery("from TravelReview");
     List<TravelReview> reviews = query.list();
     for (TravelReview travelReview : reviews) {
         System.out.println("Travel Review: " + travelReview);
     }
     …
}
```

HQL 문을 인자로 가진 query 인스턴스가 있다면 list() 메소드를 호출해서 테이블에서 모든 레코드를 조회할 수 있다. 내부를 보면 하이버네이트가 실행될 때 모든 데이터베이스 레코드를 TravelReview 인스턴스로 변환하고 이 레코드들을 java.util.List 콜렉션으로 제공한다. List는 표준 자바 콜렉션이므로 각 여행 리뷰(인스턴스)를 이전 코드에서 보여준 for 반복문을 이용하여 간단하게 가져올 수 있다.

JDBC와 앞의 동작을 비교해 보자. 하이버네이트에서는 한두 줄의 코드 만으로(메소드 체이닝을 이용하여 한 줄로 표기할 수 있다) 각 인스턴스를 쉽게 조회하지만, JDBC에서는 인스턴스를 가져오는 일이 쉽지 않다. 다음은 JDBC를 이용하여 레코드를 가져오는 코드다.

```
private void queryMovies() {
     List<Movie> movies = new ArrayList<Movie>();
     Movie m = null;
     try {
         Statement st = getConnection().createStatement();
         ResultSet rs = st.executeQuery("SELECT * FROM MOVIES");
         while (rs.next()) {
             m = new Movie();
             m.setId(rs.getInt("ID"));
             m.setTitle(rs.getString("TITLE"));
             movies.add(m);
         }
     } catch (SQLException ex) {
         System.err.println(ex.getMessage());
     }
}
```

인스턴스를 생성하고, 쿼리를 실행해서 ResultSet 인스턴스를 반환한다. ResultSet 인스턴스는 객체 형태가 아니라 레코드로 구성되어 있다. 각각의 컬럼을 얻기 위해서는 각각의 ResultSet 인스턴스를 찾아야 한다. 그러고 나서 각 컬럼에서 이름이나 위치 값으로 각 필드를 선택하여 가져온다(여기서는 컬럼명을 이용하여서 필드 정보를 가져왔다). 이렇게 추출된 컬럼 정보로 Movie 객체를 생성하고 movies 콜렉션에 추가한다. 다루기가 쉽지 않다.

하이버네이트에서는 딱 두 줄의 코드로 앞의 모든 작업을 추상화하므로 저레벨의

작업을 다룰 필요가 없다. 원한다면 메소드를 연결해서 한 줄로 만들 수도 있다. 예를 들어, Query 객체를 인스턴스로 만드는 대신 다음과 같이 바로 list() 메소드를 호출하면 된다.

```
List<TravelReview> reviews =
session.createQuery("from TravelReview").list();
```

이것이 하이버네이트의 힘이다. 단 한 줄의 코드로 Movie 객체의 콜렉션 형태를 얻을 수 있다. 하이버네이트에는 데이터 쿼리에 관한 많은 것이 담겨있다. 다음 부분에서 이런 기능을 더 살펴보자.

7.1.2 페이지네이션

몇 개의 레코드만 가져오려면 setMaxResults() 메소드에 한계치와 함께 호출함으로써 페이지네이션Pagination 기능을 사용할 수 있다.

```
Query query = session.createQuery("fromTravelReview");
query.setMaxResults(100);
query.setFirstResult(10);*
List<TravelReview> reviews = query.list();
```

setMaxResults() 메소드는 오직 100개의 레코드만 보기를 원한다고 하이버네이트에 알려준다. 쿼리에 추가 옵션이 있다. 결과 셋에 시작 위치를 지정하는 것이다. 어느 위치로부터 100개의 레코드가 보여지길 원하는가(맨 처음 또는 10번째 또는 1000번째부터)? 쿼리 인스턴스의 setFirstResult() 메소드를 사용하여 가져올 레코드의 시작 위치를 정할 수 있다. 이 코드에서는 테이블의 10번째 레코드부터 100개의 객체를 가져온다.

7.1.3 고유 레코드 조회하기

질의 조건에 따라 오직 한 개의 레코드만 있다는 것을 알 때, Query API의 uniqueResult() 메소드를 사용하면 된다. 런던 여행 리뷰를 조회한다면(런던에 관한 리뷰는 오직 한 개만 있다고 예상한다) 쿼리는 다음과 같다.

```
private void getTravelReviewUniqueRecord() {
     …
     Query query = session.createQuery("from TravelReview where
title= ' London ' ");
     TravelReview review = (TravelReview) query.uniqueResult();
}
```

uniqueResult()는 오직 한 개의 레코드가 존재한다는 것을 알 때 사용할 수 있는 편리한 메소드다.

7.1.4 네임드 매개변수

앞의 HQL 쿼리에서 London으로 title 컬럼 값을 하드 코딩했다. 그러나 입력 기준 값을 하드 코딩해서는 안 된다. 이런 질의 기준은 종종 변할 수 있기 때문이다.

예를 들어, 여행 사이트의 한 사용자 런던에 관한 리뷰를 요청한다고 해보자. 다른 사용자는 하이데라바드[01]에 관한 리뷰를 요청할 수도 있다. 두 경우 모두, 입력되는 도시명만 변경하면 되는데, 이를 매개변수화해야 한다. 특별한 문법을 사용해서 Query 객체에 입력 매개변수를 지정할 수 있다. 다음을 살펴보자.

```
private void getTravelReviewWithQueryParam(String city) {
     …
     Query query =
```

01 하이데라바드는 인도 남쪽에 위치한 도시로, 국제적으로 하이데라바디 비리야니 요리로 유명하다.

```
        session.createQuery("from TravelReview where title=:titleId");
      query.setString("titleId", city);
}
```

이 예제에서 ":titleId"는 도시명을 입력받기 위한 바인딩 변수(플레이스홀더 placeholder)다. 이 다음 줄에 입력 매개변수와 함께 바인딩 변수에 사용할 값을 지정한다. 질의 기준에 맞게 바인딩 변수를 생성한다.

```
Query query = session.createQuery(
"from TravelReview where title=:titleId and id=:reviewId");
query.setString("titleId", "London");
query.setInteger("reviewId",1);
```

7.1.5 IN 옵션 사용하기

선택된 목록 기준에 맞게 데이터를 조회해야 할 수도 있다. 예를 들어, 매개변수로 들어온 목록에 포함된 도시의 리뷰를 조회하려면 HQL의 IN 옵션을 사용한다. 이것은 SQL의 IN 연산자와 동일하다.

질의 기준에 맞는 목록이 필요한데, 이는 표준 자바 콜렉션 클래스를 이용하여 만든다. 다음 예제에서는 선택한 도시 목록을 추가하기 위해 ArrayList()를 사용한다. 그리고 Query API의 setParameterList() 메소드를 사용하는데, 이는 가져올 아이템 목록을 받는다.

```
private void getTravelReviewWithQuery ParamList() {
{
      …
      // List를 정의하고 기준을 추가한다.
      List titleList = new ArrayList(); titleList.add("London");
titleList.add("Venice");
```

```
    // Query 생성하기
    Query query = session.createQuery("from TravelReview where title in
(:titleList)");

    // titleList를 참조하는 네임드 매개변수를 어떻게 지정하는지 주목하자.
    query.setParameterList("titleList", titleList);;
    List<TravelReview> reviews =  query.list();
    …
}
```

선택 기준 목록을 만들고 나면 setParameter() 메소드를 통해 쿼리에 추가한다. 하이버네이트 런타임은 이 HQL 쿼리를 동일한 SQL 쿼리로 변환한다.

```
SELECT * from TRAVEL_REVIEW where title in( ' London ' , ' Hyderabad ' ,…)
```

7.1.6 위치 매개변수

앞에 나온 네임드 바인딩 변수(:<name>을 접두사로 가진)를 이용하는 대신 위치 바인딩 변수를 사용할 수 있다. 위치 바인딩 변수는 쿼리에서 이름/값의 문자열 바인딩을 사용하지 않고, 위치 값으로 정수값을 이용한다. 여기서는 바인딩 변수로 물음표(?)를 사용한다. 앞의 코드를 다시 작성하면 다음과 같다.

```
Query query = session.createQuery("from TravelReview where title=? and id=?");

// 0 번째 위치에 title 변수
query.setString(0, "London");

// 첫 번째 변수에 id 변수
query.setInteger(1,1);
```

위치 바인딩 변수는 0부터 시작한다.

위치 매개변수를 살펴봤다. 그러나 네임드 매개변수를 지원하기 위해 위치 변수를 사용하지 않도록 권고한다. 쿼리에 매개변수를 지정할 때에는 네임드 매개변수 표준 코딩 규칙을 사용하라.

7.1.7 별칭

질의하는 객체에 이름을 부여할 때도 있다. 테이블에 부여된 이 이름을 별칭alias이라 하며 조인이나 서브쿼리를 만들 때 매우 유용하다.

앞의 HQL 쿼리는 다음과 같이 재작성할 수 있다.

```
// tr은 객체에 부여된 별칭
Query query = session.createQuery(
     "from TravelReview as tr where tr.title=:title and tr.id=:id"
);
```

as 키워드는 선택적으로 사용한다.

7.1.8 반복자

query.list() 메소드는 리스트를 반환하고 차례로 반복자iterator를 반환한다. 반복자는 자바 콜렉션 툴킷에 포함된 것으로, 리스트를 반복하는 기능을 한다. 반복자를 얻기 위해 다음과 같이 query.list().iterator() 메소드를 호출한다.

```
Query query =  session.createQuery("from TravelReview");
Iterator queryIter =  query.list().iterator();
while(queryIter.hasNext()){
     TravelReview tr = (TravelReview)queryIter.next();
     System.out.println("Travel Review:" + tr);
}
```

콜렉션에서 아이템을 삭제하려면 반복자를 사용해야 하는데, iterator.remove() 메소드를 사용하면 된다.

7.1.9 선택 연산자

HQL에서 SELECT 연산자는 SQL에서와 비슷한 역할을 한다. 데이터베이스에서 특정 컬럼을 조회하려면 SELECT 연산자를 사용하면 된다. 대소문자를 구분하지 않으므로 SELECT는 select와 동일하다(다른 연산자도 마찬가지다). 컬럼의 리스트는 객체 타입이 없는 리스트 형태로 조회된다. 예를 들어, 테이블에서 모든 도시의 모든 컬럼이 아니라 오직 리뷰(review) 설명만 조회하는 방법은 다음과 같다.

```
// tr은 객체의 별칭이다
Query query =
     session.createQuery("SELECT tr.review from TravelReview as tr");

//각 리뷰는 문자열 형태의 긴 설명이다.
List<String> reviews = query.list();

System.out.println("City Review:");
// 모든 결과 컬럼 반복
for (String review : reviews) {
     System.out.println("\t" + review);
}
```

앞의 예제에서 한 개의 컬럼만 선택하였으므로 타입을 예상할 수 있었다. 그러나 복수 개의 컬럼을 선택할 때는 쿼리에서 어떻게 결과 셋을 반환할까? 선택한 복수 개의 컬럼은 튜플tuple 형태로 반환된다.튜플은 객체로 이루어진 단순히 객체로 구성된 배열이다. 다음 예제에서는 선택 쿼리의 title과 review 컬럼을 선택한다.

```
private void getTravelReviewWithSelectTuples() {
```

```
        …
        String SELECT_QUERY_MULTIPLE_COLUMNS =
        "SELECT tr.title, tr.review from TravelReview as tr";
        Query query = session.createQuery(SELECT_QUERY_MULTIPLE_COLUMNS);
        Iterator reviews = query.list().iterator();
        while(reviews.hasNext()){
            Object[] r = (Object[])reviews.next();
            System.out.print("Title:"+r[0]+"\t");
            System.out.println("Review:"+r[1]);
        }
    }
```

반환된 아이템 리스트는 객체로 이루어진 배열 형태로 제공된다. title과 review는 앞의 예제에서 보듯이 결과 배열의 첫 번째, 두 번째 요소로 각각 조회할 수 있다.

튜플을 예상하고 작업하는 것은 좋은 방법이 아니다. 하이버네이트 툴킷의 다른 기능을 사용하는 좀 더 효율적인 방법이 있는데, 결과를 도메인 객체로 바꾸는 것이다. 예를 들어, title과 description 속성으로 구성된 City 인스턴스가 있다고 가정해 보자. 다음과 같이 조회한 행마다 City 인스턴스를 생성해야 한다.

```
String QUERY = "SELECT new City(tr.title, tr.review ) from TravelReview as tr";
// city 리스트 얻기
List<City> cities = session.createQuery(QUERY).list();
for (City city : cities) {
     System.out.println("City: "+city);
}
```

쿼리 내에서 City 객체의 인스턴스가 생성된다. SELECT new City(…)는 로드되면서 인스턴스를 생성하는데, 이는 city 객체를 새로 생성하는 꽤 편리한 방법이다.

7.1.10 집계 함수

하이버네이트는 avg(), min(), max(), count(*)와 같은 집계 함수를 제공한다. 이 역시 SQL에서 제공하는 기능과 같다. 다음 코드는 특별한 설명이 없어도 이해할 수 있을 것이다.

```
// 가장 비싼 항공권의 가격 조회하기
List review = session.createQuery(
      "select max(ticket_price) from TravelFlight")
      .list();
// galaxy 테이블에서 행성의 평균 나이 조회하기
List review = session.createQuery(
      "select avg(planet_age) from Galaxy")
      .list();
```

max(ticket_price) 함수는 데이터베이스에 기록된 가장 높은 항공권 가격을 반환한다. 이와 유사하게 avg() 집계 함수로 행성의 평균 나이를 계산할 수 있다.

7.1.11 갱신과 삭제

이전 장에서 세션의 save()와 delete() 메소드로 엔티티를 영속화하거나 삭제했다. 데이터를 갱신하고 삭제하는 또 다른 방법이 있는데, 쿼리 문자열과 매개변수를 바인딩하는 방법이다.

```
// 레코드 갱신하기
String UPDATE_QUERY="update TravelReview set review=:review where id=2";
Query query = session.createQuery(UPDATE_QUERY);
query.setParameter("review", "The city with charm.
The city you will never forget");
int success = query.executeUpdate();
```

갱신 쿼리에서는 id=2를 가진 레코드의 리뷰 값을 지정하기 위해 바인드 매개변수를 사용한다. executeUpdate() 메소드는 성공할 경우 테이블을 갱신하고 정수값을 반환한다. 같은 메소드를 이용하여 delete 문을 실행할 수 있다.

```
// 레코드 삭제하기
String DELETE_QUERY="delete TravelReview where id=6";
Query query = session.createQuery(DELETE_QUERY);
int success = query.executeUpdate();
```

7.1.12 Criteria

앞에서 데이터 필터링을 하기 위해 WHERE 절을 이용한 SQL 기반 쿼리를 사용했다. 하이버네이트에서는 criteria를 도입하여 필터링의 또 다른 방법을 제공한다.

Criteria와 Restrictions 클래스를 이용하여 런던 여행 리뷰를 조회해 보자.

```
Criteria criteria = session.createCriteria(TravelReview.class);
List review = criteria.add(Restrictions.eq("title", "London")).list();
System.out.println("Using equals: " + review);
```

세션에서 Criteria 인스턴스를 생성하고, 그다음 Restrictions를 추가한다. Restrictions는 eq()(같다), ne()(같지 않다), like() 같은 몇 가지 정적 메소드를 가지고 있다. 앞에서 title이 London인 모든 리뷰를 조회했다. 이 결과셋을 얻기 위해 그 어떤 SQL도 사용하지 않았다.

다음과 같이 메소드 체이닝으로 creteria 인스턴스에 restrictions를 추가할 수 있다.

```
Criteria criteria = session.createCriteria(TravelReview.class)
```

```
        .add(Restrictions.eq("author", "John Jones"))
        .add(Restrictions.between("date",fromDate,toDate))
        .add(Restrictions.ne("title","New York"));
```

실제로 Criteria 클래스를 사용하지 않고, 마지막에 list() 메소드를 체이닝해서 이 부분을 간소화할 수 있다.

```
List reviews = session.createCriteria(TravelReview.class)
        .add(Restrictions.eq("author", "John Jones"))
        .add(Restrictions.between("date",fromDate,toDate))
        .add(Restrictions.ne("title","New York")).list();
```

이 코드는 객체지향 기술을 이용하여 주어진 selection criteria에 해당하는 리뷰를 조회한다.

적은 수의 컬럼을 조회한다면 Projections 클래스를 이용하면 된다. 예를 들어, 다음 코드는 테이블의 title 컬럼을 가져온다.

```
// 모든 title 컬럼을 선택한다.
List review = session.createCriteria(TravelReview.class)
        .setProjection(Projections.property("title"))
        .list();

//컬럼의 개수를 조회한다.
review = session.createCriteria(TravelReview.class)
        .setProjection(Projections.rowCount())
        .list();

// title의 개수를 조회한다.
review = session.createCriteria(TravelReview.class)
        .setProjection(Projections.count("title"))
        .list();
```

Projections 클래스는 avg()(평균), row Count()(SQL의 count(*) 과 동일하다), count()(컬럼의 개수), max()(최대값), min()(최소값)과 같은 몇 가지 정적 메소드를 가지고 있다.

7.1.13 네임드 쿼리

하드 코딩한 쿼리는 실습 예제로는 좋지 않다. 이 하드 코딩 제약을 제거할 두 가지 방법이 있다. 클래스 레벨에서 엔티티의 쿼리를 사용하기 위해 @NamedQuery 어노테이션를 이용하거나 매핑 파일에 선언할 수 있다. 두 방법 모두 네임드 쿼리는 세션에서 가져온다. 어노테이션을 이용할 때에는 TravelReview 엔티티에 @NamedQuery 어노테이션을 추가한다. @NamedQuery 어노테이션은 다음 예제처럼 name과 query 속성이 있다.

```
@Entity(name="TRAVEL_REVIEW")
@NamedQuery(name = "GET_TRAVEL_REVIEWS",
     query = "from com.madhusudhan.jh.hql.TravelReview")
public class TravelReview implements Serializable {
     …
}
```

어노테이션으로 네임드 쿼리를 정의하면 실행하는 동안 세션에서 가져올 수 있다. 다음 예제처럼 특정 네임드 쿼리를 참조하려면 name 속성을 반드시 사용해야 한다.

```
private void usingNamedQueries() {
     …
     // 선언된 네임드 쿼리를 가져온다
     Query query = session.getNamedQuery("GET_TRAVEL_REVIEWS");
     List reviews = query.list();
}
```

다음 예제와 같이 부모 @NamedQueries 어노테이션에 각 @NamedQuery를 추가하여 한 개의 엔티티에 여러 개의 쿼리를 연결할 수 있다. @NamedQueries는 value 인자를 받는데, 이는 @NamedQuery를 정의하는 배열로 만들어진다.

```
@Entity(name = "TRAVEL_REVIEW")
@NamedQueries(
     value = {
         @NamedQuery(name = "GET_TRAVEL_REVIEWS", query = "from TravelReview"),
         @NamedQuery(name = "GET_TRAVEL_REVIEWS_FOR_TITLE",
             query = "from TravelReview where id=:title")
     }
)
public class TravelReview implements Serializable { … }
```

명시적인 방법을 사용할 때는 매핑 파일에 쿼리를 정의해야 한다. 예를 들어, TravelReview.hbm.xml 파일에 travel 관련 쿼리를 정의한다. 쿼리는 엔티티 클래스와 관련된 hbm 파일에 추가해야 한다.

```
<hibernate-mapping>
     <class name="com.madhusudhan.jh.hql.TravelReview"
table="TRAVEL_REVIEWS">
     …
     </class>
     <!-- 엔티티 관련 쿼리를 이곳에 정의한다 -->
     <query name="GET_TRAVEL_REVIEWS">
         <![CDATA[ from TravelReview ]]>
     </query
     <query name="GET_TRAVEL_REVIEWS_FOR_TITLE">
<![CDATA[ from TravelReview where id=:title ]]>
     </query
</hibernate-mapping>
```

7.2 네이티브 SQL

하이버네이트에서는 네이티브 SQL 쿼리를 실행하는 기능도 제공한다. session.createSQLQuery() 메소드는 SQLQuery 객체를 반환한다. Query 객체를 반환하는 createQuery() 메소드와 비슷한데, 이 클래스는 앞에서 살펴본 Query 클래스를 확장한 것이다.

HQL이 사용하기 간편한데 왜 네이티브 SQL을 써야 할까? 데이터베이스 벤더의 특정 함수 또는 생성에 의존적인 쿼리문이 있을 때 다음과 같이 네이티브 SQL 전략을 사용한다.

```
SQLQuery query = session.createSQLQuery("select * from NATIVESQL_EMPLOYEE");
List employees = query.list();
```

쿼리 문자열은 실행이 가능한 SQL 문이다. HQL에서는 쿼리 시작 부분에 있는 SELECT 키워드를 생략할 수 있다.

앞에서 보았듯이 코드 기반 외부에 네이티브 SQL 쿼리를 선언하거나 정의할 수 있다. 엔티티 자체에 @NameQuery 어노테이션을 정의하거나 매핑 파일에서 sql-query 요소를 선언해서 네임드 쿼리를 이용한다. 다음 예제를 살펴보자.

```
<hibernate-mapping>
    <class name="com.madhusudhan.jh.hql.TravelReview"
table="TRAVEL_REVIEWS">
    ...
    </class>
    <!-- 엔티티와 관련된 쿼리를 이곳에 정의하자 ->
    <sql-query name="GET_TRAVEL_REVIEWS">
       <![CDATA[ SELECT * from TravelReview ]]>
    </sql-query
```

```
</hibernate-mapping>
```

프로그램 내에서 네임드 쿼리를 조회하려면 세션의 `getNamedQuery()` 메소드를 사용한다. 네임드 쿼리에 참조할 정보를 넘기면(앞의 예제에서는 `"GET_TRAVEL_REVIEWS"`를 쓴다). `Query` 객체가 반환된다.

7.3 요약

이 장에서는 HQL의 기본 동작과 주요 기능을 살펴봤다. HQL은 하이버네이트에 추가된 강력한 도구다. 그리고 HQL과 SQL을 비교하여 알아보고, `Criteria`와 `Projections`에 대해 살펴봤다.

8 | 자바 퍼시스턴스 API

자바 퍼시스턴스 API[JPA, Java Persistence API]는 객체 영속성 프로그래밍을 위한 표준을 만들기 위해서 개발되었다. ORM 퍼시스턴스 벤더에 배포한 가이드라인이 매우 잘 적용된 사례들에 따르면, JPA 명세의 핵심 부분은 하이버네이트로부터 비롯되었다. 그래서 구현된 명세들이 새롭게 느껴지지 않을 것이다.

이 장의 목적은 하이버네이트를 이용하여 JPA 규격에 맞는 애플리케이션 개발에 대한 정보를 상세히 다루는 것이다.

표준 명세는 소프트웨어 컴포넌트를 독립된 환경에 좀 더 쉽게 맞추고 재사용할 수 있게 하고, 많은 문제를 발생시키지 않고 프레임워크를 변경할 수 있도록 도와준다. 자바 커뮤니티에서는 JCP 프로그램을 통해 표준 명세를 도입함으로써 표준의 확산에 많은 일을 했다.

객체-관계 간의 영속성 세상의 초기에는 ORM 프레임워크를 위한 어떤 명세도 없어서 소프트웨어를 스스로 만들어 쓰거나 서드 파티의 비표준 제품을 사용했다. 이는 벤더에 종속적으로 만들었는데, 이들 프레임워크는 유연하지도 견고하지 않았다. 이는 해당 데이터베이스에 얽혀있는 아키텍처를 만들어내는 것을 의미했다. 자신의 프로그램을 만들든 공급업체의 상품을 사용하든 모두 큰 문제였다. 몇몇 골치 아픈 경우에서는 프로젝트 구조를 변경하거나 원점부터 시작하지 않고는 상품이나 데이터베이스에 맞추지 못했다.

표준 영속성의 필요성을 깨닫게 되면서 자바팀은 JPA를 만들었다. 표준화되고 일관된 자바 영속성 소프트웨어 모듈을 만드는 프레임워크을 얻기 위해 계속 JPA를 유지해 나갔다. 하이버네이트는 좋은 영속성 ORM 툴을 만드는 명세를 채택한 프

레임워크의 하나다. 오히려 이제는 하이버네이트가 검증된 설계로 표준화에 기여하고 있다.

이 장에서는 하이버네이트가 JPA에 기여한 부분을 자세히 살펴보겠다.

8.1 하이버네이트와 JPA

JPA는 명세이고, 하이버네이트는 이 명세에 적힌 규칙을 따르는 구현체의 공급자다. 이미 Session, SessionFactory 같은 하이버네이트 API 클래스에 익숙할 것이다. 이런 클래스들은 org.hibernate 패키지에 속한 하이버네이트 클래스들이며, 라이브러리에 포함되어 있다.

하이버네이트는 JPA 명세를 따를 뿐 아니라 EntityManager와 EntityManagerFactory 클래스와 대응되는 하이버네이트 API도 만들었다. 하이버네이트 API와 달리 이 클래스들은 javax.persistence 패키지에 포함되었다. 따라서 JPA 클래스에 맞게 애플리케이션을 개발하면 하이버네이트에서 EclipseLink와 같은 다른 JPA 공급자로 변경하는 일을 쉽게 처리할 수 있다.

언제나 표준 명세를 사용하자

시스템을 설계할 때 가능한 한 벤더 제약 없는 구조를 고려하자. 기술적인 아키텍처가 벤더의 상품과 그 API로 복잡해지는 것을 수없이 봐 왔다. 이는 종종 빠져나오기 힘든 상황과 얽히게 된다. 특정 벤더에서 제공하는 제품의 기능을 염두에 두고 시스템 만들어야 한다면 필요한 인터페이스를 설계하고 작업의 범위를 정하는 것이 최우선일 것이다. 즉, 벤더를 고려하여 솔루션을 설계하지 말아야 한다. 그대신 자신의 솔루션을 지원한 제품과 프레임워크로 기술적인 아키텍처를 보완해야 한다. 하이버네이트에서 다른 프레임워크로 옮길 때 많은 작업을 피하려면 하이버네이트 자체 API보다 JPA를 사용하자.

크게 보면 JPA 기반 시스템 작업에는 세 가지 기본적인 부분이 있다. 다음에서 이것들을 살펴보겠다.

8.1.1 영속성 컨텍스트

영속성 컨텍스트Persistence Context는 영속화 객체의 집합이다. 예를 들어, 특정 데이터베이스에 맞는 Instrument, Trade, Security와 같은 많은 엔티티를 가지고 있다고 해보자. 다음처럼 trading_entities라는 이름으로 영속성 컨텍스트를 생성하기 위해서 모든 엔티티를 모아둔다.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence
      xmlns="http://java.sun.com/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
          http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
      version="2.0">
      <persistence-unit name="trading_entities"
          transaction-type="RESOURCE_LOCAL">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
          <class>com.madhusudhan.jh.jpa.Instrument</class>
          <class>com.madhusudhan.jh.jpa.Trade</class>
          <class>com.madhusudhan.jh.jpa.Security</class>
          <class>com.madhusudhan.jh.jpa.Risk</class>
          <properties>
              <property name="hibernate.connection.url"
                  value="jdbc:mysql://localhost:3307/JH"/>
              <property name="hibernate.connection.driver_class"
                  value="com.mysql.jdbc.Driver"/>
              <property name="hibernate.dialect"
                  value="org.hibernate.dialect.MySQL5Dialect"/>
              <property name="hibernate.connection.username"
                  value="myusername"/>
              <property name="hibernate.connection.password"
```

```
                value="mypassword"/>
            </properties>
        </persistence-unit>
    </persistence>
```

먼저, trading_entities라는 이름으로 persistence-unit 명을 정의한다. 그리고 실행 시에 JPA 구현체 공급자(하이버네이트) 정보를 알리기 위해 provider 태그에 org.hibernate.ejb.HibernatePersistence 값을 지정한다. 그러고 나서 모든 영속 클래스를 차례로 정의한다. 또한, 데이터베이스 연결 정보를 포함하여 정의된 속성이 있다. 이는 데이터베이스에 연결하거나 접근할 때 공급자가 사용한다.

주의할 점은 META_INF 폴더의 persistence.xml 파일에 설정해야 한다는 점이다. 이 폴더가 없다면 프로젝트 레벨에서 이 폴더를 생성해야 한다. 또한 META-INF 디렉터리가 애플리케이션의 클래스패스에 추가되었는지 확인해야 한다. 하이버네이트 JPA 런타임은 영속성 컨텍스트의 생성과 로딩을 위한 persistece.xml 파일을 찾기 위해서 META-INF 디렉터리를 검색한다.

persistece-unit 내에 존재하는 영속화 엔티티는 이전 장에서 본 어노테이션을 이용하여 엔티티를 생성한다. 관련 내용을 다시 확인하기 위해 어노테이션과 정의된 Trade 엔티티를 살펴보자.

```
@Entity
@Table(name="TRADE")
public class Trade {
        @Id
        @Column(name="TRADE_ID")
        private int id = 0;
        ...
}
```

영속성 정의 파일에는 복수 개의 persistece unit을 정의할 수 있다. 여러 개의 데이터베이스로 작업해야 할 때에 특히 유용하다. 예를 들어, Trade 관련 unit은 MySQL, Report 엔티티는 오라클에 영속화할 수도 있다. 다음 예제에서는 두 개의 persistence unit과 각각 사용할 데이터베이스를 위한 속성을 정의한다.

```
<persistence>
<!-- MySQL 데이터베이스를 위한 Persistence unit 정의 -->
      <persistence-unit name="trading_entities"
          transaction-type="RESOURCE_LOCAL">
          <provider>org.hibernate.ejb.HibernatePersistence</provider>
              <class>com.madhusudhan.jh.jpa.Instrument</class>
              <class>com.madhusudhan.jh.jpa.Trade</class>
              <properties>
                  <property name="hibernate.connection.url"
                      value="jdbc:mysql://localhost:3307/JH"/>
                  <property name="hibernate.connection.driver_class"
                      value="com.mysql.jdbc.Driver"/>
                  ...
      </persistence-unit>

      <!--  Oracle 데이터베이스를 위한 Persistence unit 정의 -->
      <persistence-unit name="reporting_entities"
          transaction-type="RESOURCE_LOCAL">
          <provider>org.hibernate.ejb.HibernatePersistence</provider>
              <class>com.madhusudhan.jh.jpa.report.Entity</class>
              <properties>
                  <property name="hibernate.connection.url"
                      value="jdbc:oracle:thin:@localhost:1521:JH"/>
                  <property name="hibernate.connection.driver_class"
                      value="oracle.jdbc.driver.OracleDriver"/>
                  ...
```

```
        </persistence-unit>
    </persistence>
```

한 개의 설정 파일에 MySQL과 오라클에 해당하는 persistence unit을 각각 설정한다.

persistence unit 정의에서 어떤 일들이 일어나고, 이 정의는 어디서 로딩될까? Persistence라는 유틸리티 클래스가 있다. 이 클래스는 클래스패스에 있는 META-INF 폴더의 persistece.xml 파일을 찾는다. 런타임 때 persistence.xml 파일이 로드되고, 파일에 정의된 persistence unit들을 인스턴스화한다.

persistence unit을 정의했으니 다음으로 EntityManagerFactory와 EntityManager 클래스를 살펴보자.

8.1.2 EntityManagerFactory

EntityManagerFactory는 EntityManger 클래스를 생성하는 팩토리 클래스다. 이 클래스는 다소 무겁고 Thread-safe하게 구현된 객체다. 이런 이유로 각 persistence unit은 오직 하나의 EntityManagerFactory 인스턴스에 의해서 생성된다. EntityManagerFactory를 생성하는 것은 비용이 많이 들므로 인스턴스를 캐시하여 접근하는 방법을 추천한다. 하지만 EntitymanagerFactory에서 생성된 EntityManager는 사용하기에 안전하고, 사용되지 않으면 버려진다. 한 개의 persistence unit에 한 개의 EntityManagerFactory가 생성된다. 그러나 단일 JVM에서는 단일 persistence unit에 맞는 복수 개의 EntityMangerFactory를 가질 수 있다. 다음처럼 persistence unit명을 넘겨서 생성된 Persistence 인스턴스로부터 팩토리를 얻을 수 있다.

```
// trade 엔티티와 관련된 persistece unit
```

```
EntityManagerFactory tradeFactory =
      Persistence.createEntityManagerFactory("trading-entities");
// report 엔티티와 관련된 persistece unit
EntityManagerFactory reportFactory =
      Persistence.createEntityManagerFactory("report-entities");
```

Trade 엔티티와 Report 엔티티의 persistece unit을 위한 두 개의 EntityManagerFactory는 이전 코드 부분마다 생성된다. 그리고 EntityManager 인스턴스 생성을 위해 이 팩토리를 사용한다. 이는 데이터베이스로의 게이트웨이 역할을 한다.

EntityManagerFactory 인스턴스를 얻은 후에는 EntityManager를 인스턴스화하고 이는 다음에서 살펴본다.

8.1.3 EntityManager

모든 관점에서 EntityManager는 Session과 같다. 이는 엔티티의 생명주기를 관리한다. 단일 작업 단위를 가지고 엔티티 클래스를 작업할 데이터베이스에 상호연동한다. 주어진 트랜잭션을 위해 현재 실행 스레드와 묶이고, 1차 캐시를 유지한다. EntityManager에 연결된 활성화된 트랜잭션이 완료되면 캐시는 재사용된다. EntityManager는 항상 영속성 컨텍스트와 관계를 맺고 있다.

EntityManager에는 두 가지 타입이 있다. 하나는 컨테이너 관리 환경에서 실행되고, 다른 하나는 standalone JVM에서 동작한다. 전자는 일반적으로 애플리케이션 서버나 웹 컨테이너와 같은 J2EE 컨테이너고, 후자는 JSE 프로그램이다.

두 타입에서 EntityManger는 차이가 없지만 EntityManager 생성을 책임지는 EntityManagerFactory에는 차이가 있다. 컨테이너 관리 환경에서는 설정이 시작되면 EntityManagerFacory가 인스턴스화가 되고 DI[dependency injection]를 통해 어

플리케이션에서 사용된다. standalone 모드에서 설정과 팩토리를 생성하는 것은 애플리케이션의 몫이다. 다음은 Standalone 모드에서 entityManagerFactory와 EntityManager를 생성하는 방법을 보여준다.

```
EntityManagerFactory factory =
      Persistence.createEntityManagerFactory("trading-entities");
EntityManager manager = factory.createEntityManager();
```

컨테이너 관리 환경에서 EntityManager는 컨테이너가 주입한다. Persistence unit을 찾고 entityManagerFactory를 생성하고, EntityManager를 생성하고 주입하는 책임은 J2EE 애플리케이션 컨테이너에 있다.

다음과 같이 @Resource 어노테이션이 붙은 엔티티의 EntityManager 인스턴스는 주입될 준비된 상태다.

```
@Resource
private EntityManager manager = null;
```

8.2 영속성 오브젝트

하이버네이트 영속성 모델과 비슷하게 JPA는 개발자들이 영속화 작업을 할 수 있도록 자체 API를 제공한다. EntityManager는 중요한 클래스며, 데이터베이스에 연결된 주요 문이라고 생각하면 된다

8.3 엔티티 저장하고 질의하기

데이터베이스에 엔티티를 영속하기 위해 세션의 save()와 saveOrIpdate() 메소드를 이용한다. 이 메소드와 마찬가지로 JPA 모드에서는 EntityManager의 persistEntity() 메소드 이용하여 객체를 데이터베이스에 저장할 수 있다

다음 예제는 이 내용을 보여준다.

```
public void persistNewInstrument(){
// EntityManager 생성하기
EntityManager manager = entityManagerFactory.createEntityManager();
// instrument 객체를 생성하고 populate하기
Instrument instrument = new Instrument();
instrument.setIssue("IBM");
// 도메인 객체 저장하기
manager.persist(instrument);
```

다음처럼 getReference()메소드를 통해서 객체를 찾고 가져온다.

```
public void findInstrument() {
      Instrument instrument = manager.getReference(Instrument.class,1);
}
```

getReference() 메소드에서는 두 개의 매개변수인 도메인 클래스 자신과 기본키를 갖는다.

객체를 가져오는 방법에는 getReference() 메소드와 비슷한 find() 메소드도 있다. 이 메소드는 다음과 같이 사용한다.

```
public void findInstrument() {
      Instrument instrument = manager.find(Instrument.class,1);
}
```

그러나 두 메소드 사이에는 약간 다른 점이 있다. getReference() 메소드는 지연 로딩lazy-loaded된 엔티티를 가져온다. 이는 기본키를 제외한 클래스의 다른 속성은 가져오지 못해서 그 속성에 접근할 때 null 값이 된다. 하지만 find() 메소드는 제

대로 가져온다. 또한, getReference() 메소드는 데이터베이스에 레코드가 없을 때 EntityNotFoundException을 반환하고, find() 메소드는 null 값만 반환한다.

엔티티 삭제는 아주 간단하다. EntityManager의 remove() 메소드를 사용하면 저장소의 엔티티를 삭제할 수 있다.

```
public void deleteInstrument() {
      manager.remove("IBM");
}
```

데이터베이스의 영속화 객체의 상태를 동기화하려면 flush()나 refresh() 메소드를 사용한다. flush() 메소드는 객체의 변경된 복사본을 가진 데이터베이스를 갱신하지만, refresh() 메소드는 반대로 데이터에서 읽어온 레코드의 최신 복사본을 객체 모델에 갱신한다.

이것으로 JPA와 하이버네이트의 고급 개념을 마무리한다.

8.4 요약

JPA는 자바의 영속성 세상를 위한 표준 명세다. 이 장에서는 JPA의 기본 개념과 API를 간단히 살펴보았다. 하이버네이트에서는 JPA를 모두 지원한다는 것과 프레임워크로부터 유연한 코드를 만들어야 할 때 JPA API를 연계해서 사용할 수 있다는 점도 살펴보았다.