Hanbit eBook

Realtime 25

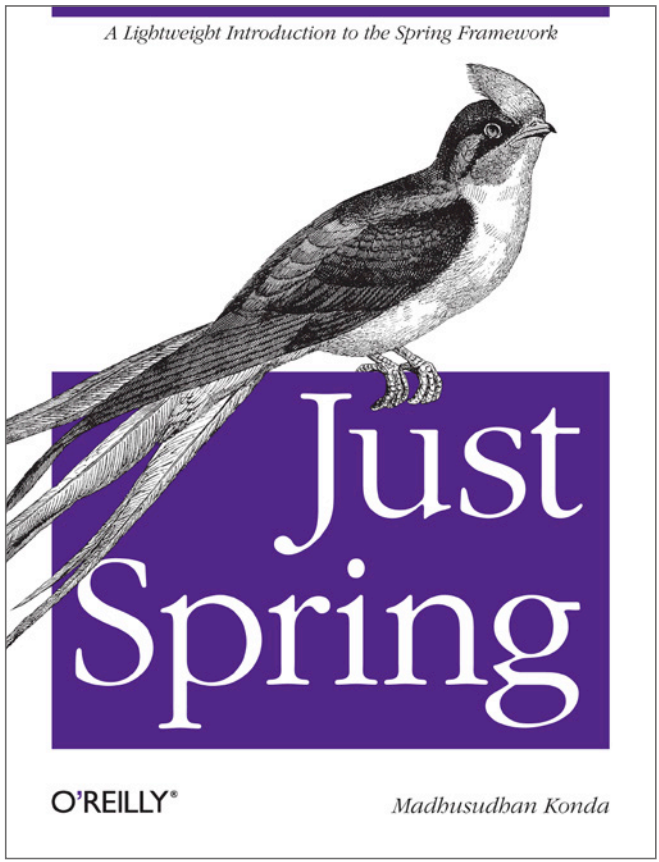
스프링 핵심 노트

빈과 컨테이너 중심으로 빠르게 배우는

Just Spring

맷허서드한 콘다 지음 / 송지연 옮김

O'REILLY® 한빛미디어 Hanbit Media, Inc.



이 도서는 O'REILLY의
Just Spring의
번역서입니다.

Just Spring 표지의 동물은 도가머리 칼새입니다.

빈과 컨테이너 중심으로 빠르게 배우는

스프링 핵심 노트

빈과 컨테이너 중심으로 빠르게 배우는 **스프링 핵심 노트**

초판발행 2013년 5월 10일

지은이 마드후수단 콘다 / **옮긴이** 송지연 / **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 978-89-6848-610-4 15000 / **정가** 9,900원

책임편집 배용석 / **기획** 이중민 / **편집** 김연숙
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 박진희
마케팅 박상용, 박주훈, 정민하

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

Published by HANBIT Media, Inc. Printed in Korea

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

지은이_ **마드후수단 콘다**

마드후수단 콘다Madhusudhan Konda는 자바 전문 컨설턴트로, 주로 투자 은행과 금융 기관에서 근무했다. 엔터프라이즈와 코어core 자바 분야에서 약 12년간 일했으며 분산, 멀티스레드, N-티어 스케일러블N-Tier Scalable과 같은 확장 아키텍처가 관심 분야다. 또한 금융 관련 고빈도high-frequency와 저지연low-latency 애플리케이션 아키텍처를 설계한 경험도 있다. 집필에 남다른 애착을 갖고 있으며 멘토링에도 관심이 많다.

옮긴이_ **송지연**

지엔텔, 노키아 지멘스 네트웍스에서 근무한 경험이 있는 WCDMA, LTE 분야의 통신 기술 엔지니어. 취미로 팀을 이루어 아이폰 애플리케이션 개발에 한동안 푹 빠져 있기도 했다. 현재는 주전공인 SW 개발 분야로 돌아와 오라클 자바 개발 팀에서 근무 중이다.

저자 서문

스프링 프레임워크는 매우 훌륭한 기술로 업계에서 많은 관심과 사랑을 받고 있으며, 두터운 사용자층을 확보하고 있습니다. 이미 이 기술에 관한 많은 책, 매뉴얼, 기사, 튜토리얼, 비디오가 범람하는 상황에서 책을 쓴다는 일은 사실 모험입니다.

그런데도 이 책을 쓰게 된 동기는 무엇이었을까요?

저는 종종 기초에 약한 프로그래머나 개발자를 만나곤 하는데, 그들 중 몇몇은 스프링을 사용해본 경험이 전혀 없는데도 불구하고 갑자기 스프링 프로젝트에 투입되어 밤새워 일하는 경우도 있었습니다. 이렇게 밤을 지새우는 사람 중에는 프로젝트와 관련해 어떤 교육도 받지 못해 어려움을 겪으면서도 동료에게 도움을 구하는 데 익숙치 못한 사람들도 있었습니다.

그런가 하면 스프링을 진정으로 공부해보고 싶어하지만 방대한 양의 매뉴얼이나 글에 눌려 포기하는 사람들도 있습니다. 이들은 매우 열정적이며 당장에라도 뭔가 성과를 내고 싶어 하지만 모든 내용을 다 읽고 소화할 수 있는 인내심이나 시간이 턱없이 부족한 게 현실입니다. 그들에게 주어진 현실이란 책을 대충 훑어보고 바로 실전에 임하는 것이죠. 그런데 기술이라는 것이 언제나 그렇듯 어느 정도 손에 익으면 더 많은 것을 알고 싶어지기 마련입니다.

저는 무언가를 처음 배우려고 할 때는 기초를 살펴본 후(목차 마지막 내용까지 빨리 살펴보고 싶은 마음이지만) 예제 코드를 직접 실행해보고, 더 나아가 직접 간단한 코드를 구현하는 방식 등으로 배우는 것을 좋아합니다. 이렇게 기술을 '느끼게 되면', 그때부터는 더 많은 호기심을 갖게 되고 이를 해결하기 위한 갈증으로 심화 과정이 담긴 두꺼운 책이나 매뉴얼, 스펙을 찾아보기 시작합니다.

이것이 바로 『스프링 핵심 요약』 시리즈에 녹아있는, 단순하지만 강력한 동기죠. 제가 이 책을 쓰는 목적은 스프링 프로젝트에 대한 간결하면서도 핵심적인 내용만 뽑아내어 쓸모 있는 내용만 담은 예제 중심의 책을 만드는 것입니다.

이 책은 스프링 프레임워크의 근본을 설명하는 책으로(원서는 『Just Spring Core』로 제목이 바뀔 예정입니다), 다른 내용 없이 오로지 스프링만 배우고자 한다면 이 책이 좋은 동반자가 되어 것입니다.

또한 스프링을 빠르고 정확하게 알고 싶거나, 한두 시간 만에 스프링을 복습하고 싶을 때 또는 주말이나 휴일 같은 짧은 시간 안에 스프링을 학습하는 경우라면 반드시 이 책을 선택하기 바랍니다. 스프링에 관한 내용 중 스프링 WebMVC와 스프링 인티그레이션 같은 부분은 이 책에서 자세히 다루지 않고 다른 『스프링 핵심 요약』 시리즈에서 다룰 것입니다.

이 책을 사랑해주신 모든 분께 진심으로 감사하다는 말을 전하고 싶습니다. 제가 그동안 받았던 긍정적인 답변, 리뷰, 코멘트, 의견, 칭찬은 제게 더 큰 용기와 힘을 낼 수 있도록 많은 도움을 주었습니다. 제가 이 책을 쓰면서 즐긴 만큼 독자분들도 똑같이 즐기실 수 있기를 바라며, 혹시라도 저나 제 책에서 마음에 들지 않는 부분이 있더라도 부디 저를 계속 사랑해주셨으면 합니다.

다른 『스프링 핵심 요약』 시리즈도 이미 출간되어 있으니(한국에서는 곧 출간할 예정입니다) 계속 지켜봐 주세요.

끝으로, 저를 믿어주고 제가 방향을 잃었을 때 바로잡아준 편집자 마이크 루키디스 Mike Loukides에게 진심으로 감사드리고 싶습니다.

모든 오라일리 팀, 특히 이 책을 집필하는 데 도움을 준 블랜새트Blanchette, 홀리 바우어Holly Bauer, 사라 슈나이더Sarah Schneider, 크리스토퍼 허스Christopher Hearse, 댄 포스미스Dan Fauxsmith에게도 진심으로 감사드립니다.

또한 이 책을 쓰는 동안 많이 인내하고 협조해준, 사랑하는 아내 제네티Jeannette와 책 쓰는 것을 이해하고 허락해줘 아빠와 보낼 시간을 희생해준 멋진 6살 아들 조슈아Joshua에게도 진심으로 고맙다는 말을 전하고 싶습니다. 그리고 많은 협조와 사랑을 보내준 인도의 가족에게도 매우 감사합니다.

사랑하는 아버지를 기리며!

마드후수단 콘다Madhusudhan Konda

www.madhusudhan.com

On Twitter @mkonda007

역자 서문

스프링은 등장한 이래로 자바에 기반을 둔 웹 개발 프레임워크로 큰 인기를 얻고 있습니다. 하지만 새로운 프레임워크를 공부하기란 쉽지 않은 일입니다. 이 책은 스프링의 개념을 상당히 간단하면서도 쉽게 설명하고 있으며, 양 또한 많지 않습니다. 스프링 프레임워크에 대한 근본적이면서도 반드시 필요한 부분을 명확하게 짚어주고 있으므로 잠시 시간을 내어 읽어본다면 많은 도움이 될 것이라 확신합니다.

자바 엔터프라이즈 개발을 고려하거나 개발이 필요하다면, 스프링은 이제 반드시 익혀야 할 기술이라고 생각합니다. 하지만 막연히 새로운 기술이라는 이유로 거부감이 든다면 이 책을 통해 일단 스프링의 기본을 습득해보는 일도 좋을 것이라 생각합니다. 이 책은 이제 개발에 뛰어든 초보 개발자들도 이해하기 쉬울 만큼, 많은 예제를 통해 친절하게 설명하며 스프링 프레임워크의 전체 내용보다는 기본 개념을 설명하므로 모든 독자가 읽을 수 있을 것으로 생각합니다.

스프링은 스프링 프레임워크 그 자체로도 훌륭한 기술이지만 다른 표준 기술, 예를 들어 하이버네이트나 JDO, JPA 등도 지원한다는 점 때문에 앞으로의 확장 가능성도 더 엿볼 수 있습니다. 따라서 스프링 프레임워크의 더 자세한 내용을 알고 싶다면 앞으로 출간할 예정인 『스프링 인티그레이션 핵심 요약』, 『스프링 데이터 핵심 요약』에도 관심을 가져주시기를 바랍니다.

개인적으로도 이 책을 번역하면서 스프링, 하이버네이트 등에 대한 자료와 API를 살펴볼 기회를 갖게 되어 많은 도움이 되었습니다. 이 때문에 매너리즘에 빠지려는 순간에 새로운 자극제가 되었으며, 예전에 프로그래밍을 처음 공부하면서 가졌던 열정을 다시금 불태울 수 있는 계기가 될 수 있었던 것 같습니다.

책이 완성되기까지 많은 분의 도움을 받았습니다. 일단 이 책을 번역할 기회를 만들어준 오빠에게 감사하며, 함께 작업하면서 많은 도움을 주신 한빛미디어 관계자 분께도 감사드립니다. 또한 역자의 부족한 지식 때문에 여러 번 귀찮게 했던 삼성 SDS 정혜미 선임에게도 깊은 감사를 드리며, 항상 제 곁에서 버팀목이 되어주신 부모님과 친구들에게도 감사를 드립니다.

번역을 마치며

역자 송지연

대상 독자 및 시리즈 도서 구성

초급 **초중급** 중급 중고급 고급

이 책은 자바 웹 애플리케이션 개발 프레임워크의 대세로 자리를 잡아 가는 스프링의 핵심 개념을 소개하는 책입니다. 자바와 XML의 기본을 알고 있으며 스프링에 관심이 있는 분이라면 누구나 읽을 수 있습니다. 또한 세 권의 시리즈로 기획되었으며 다음 내용을 다룹니다.

『스프링 핵심 노트』

스프링 프레임워크의 기본과 핵심 개념을 다룹니다. 스프링 프레임워크를 간단히 소개하고 스프링의 근간이 되는 빈과 컨테이너를 살펴봅니다. 마지막에는 스프링 JMS와 JDBC, 하이버네이트 등 데이터 처리와 관련한 내용을 소개합니다.

『스프링 인티그레이션 핵심 노트』

『스프링 핵심 요약』 혹은 스프링 프레임워크 입문서를 읽어본 독자들이 기업용 애플리케이션 통합Enterprise Application Integraion 과정을 살펴볼 수 있도록 만들어진 책입니다. 또한 개발 과정에서 비즈니스 로직을 수립하는 방법도 배울 수 있습니다.

『스프링 데이터 핵심 노트』

JDBC, 하이네이트, JPA, JDO 등 스프링 프레임워크와 연결해서 사용하는 데이터 처리 방법을 소개합니다. 스프링 프레임워크와 데이터베이스의 관계를 깊게 이해함으로써 자바 기반의 웹 애플리케이션을 자유자재로 만들 수 있는 기반을 다질 수 있습니다.

예제 파일

- 스프링 프레임워크 공식 사이트

  : http://www.springsource.org/spring-framework

- 예제 파일 다운로드 사이트

  : http://examples.oreilly.com/0636920020394/

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook 입니다.

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 내시는 선배, 전문가, 고수분에게는 보다 쉽게 집필하실 기회가 되리라 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 일정한 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 보완되고 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여, DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT기기에서 자유롭게 활용하실 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해, 독자 여러분이 언제 어디서 어떤 기기를 사용하시더라도 편리하게 전자책을 보실 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 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

01 스프링 기초 1

1.1 시작하면서 1
1.2 객체의 결합도 문제 1
1.2.1 인터페이스를 이용해 설계하기 4
1.3 스프링 소개 8
1.3.1 의존성 주입 8
1.3.2 스프링으로 IReader 인터페이스 리팩토링하기 9
1.3.3 ReaderService 클래스 13
1.4 요약 16

02 스프링 빈 17

2.1 빈 환경 설정 17
2.1.1 XML을 이용한 설정 17
2.1.2 어노테이션 이용 23
2.1.3 XML 네임스페이스 27
2.2 빈 생성하기 29
2.3 라이프 사이클 30
2.3.1 의존성 없는 빈을 인스턴스화하기 32
2.3.2 의존성 있는 빈을 인스턴스화하기 33
2.4 빈 별칭 34
2.5 익명의 빈 34
2.6 주입 방식 35
2.6.1 생성자를 이용한 주입 35

2.6.2 setter 메서드를 이용한 주입 38
2.6.3 생성자와 setter 메서드의 혼용 40
2.7 빈 콜백 40
2.7.1 init-method 속성 41
2.7.2 destroy-method 속성 42
2.7.3 공통 콜백 43
2.8 요약 44

03 컨테이너 45

3.1 컨테이너 45
3.1.1 빈 팩토리 컨테이너 46
3.1.2 애플리케이션 컨텍스트 컨테이너 47
3.2 빈 인스턴스화 50
3.2.1 정적 메서드 이용 50
3.2.2 팩토리 메서드 이용 52
3.3 콜백의 초기화와 파괴 54
3.3.1 InitializingBean 인터페이스의 afterPropertiesSet 메서드 54
3.3.2 DisposableBean 인터페이스의 destroy 메서드 55
3.3.3 선언형 혹은 프로그램형 콜백 56
3.4 이벤트 핸들링 57
3.4.1 컨텍스트 이벤트 수신 58
3.4.2 사용자화 이벤트 발생 60
3.4.3 싱글 스레드 이벤트 모델 63
3.5 오토와이어링 64

3.5.1 byName 오토와이어링 65
3.5.2 byType 오토와이어링 67
3.5.3 constructor 오토와이어링 68
3.5.4 명시적 와이어링과 오토와이어링의 혼용 68

04 심화 개념 70

4.1 빈 스코프 70
4.1.1 싱글턴 스코프 70
4.1.2 프로토타입 스코프 71
4.2 프로퍼티 파일 74
4.3 프로퍼티 에디터 77
4.3.1 자바 컬렉션 주입하기 77
4.3.2 자바 프로퍼티 주입하기 81
4.4 빈 후처리기 82
4.5 부모-자식 빈 정의 86
4.6 요약 86

05 스프링 JMS 88

5.1 JMS 간단 소개 88
5.1.1 메시징 모델 89
5.2 스프링 JMS 90
5.2.1 모든 것의 모태가 되는 JmsTemplate 클래스 90
5.2.2 메시지 발행하기 95

5.2.3 기본 설정 목적지로 메시지 전송하기 99
5.2.4 토픽 선언 101
5.2.5 메시지 수신하기 101
5.2.6 동기 메시지 수신하기 102
5.2.7 비동기 메시지 수신하기 104
5.2.8 스프링 메시지 컨테이너 106
5.2.9 메시지 컨버터 106
5.3 요약 110

06 **스프링 데이터** **111**

6.1 JDBC와 하이버네이트 112
6.1.1 스프링 JDBC 113
6.1.2 하이버네이트 122
6.2 요약 129

1 | 스프링 기초

1.1 시작하면서

스프링 프레임워크(이하 스프링)는 몇 년에 걸쳐 매우 두터운 사용자층을 형성해 왔을 뿐만 아니라 소프트웨어 개발 업체와 기업이 꼭 필요로 하는 부분을 충족시켜 주는 가장 좋은 프레임워크로 자리매김하고 있다.

놀랍게도 스프링의 핵심 원칙인 의존성 주입Dependency Injection은 매우 간단하고 이해하기도 쉽다. 1장에서는 여러분이 의존성 주입의 원칙을 이해할 수 있도록 스프링의 기본에 대해 개괄적으로 이야기할 것이다. 먼저 객체 결합과 강한 의존성tight dependency이 지닌 문제점을 이야기하고, 스프링에서 비결합성과 의존성을 어떻게 풀어나갈지 설명하겠다.

1.2 객체의 결합도 문제

다양한 데이터 소스로부터 데이터를 읽는 간단한 프로그램을 생각해보자. 파일 시스템이나 데이터베이스, 심지어 FTP 서버에서도 데이터를 읽어올 수 있어야 한다. 여기서는 파일 시스템으로부터 데이터를 읽는 간단한 프로그램을 만든다. 최적의 사례나 의존성 주입 패턴을 이용하지 않은, 단순히 동작만 하는 프로그램이다.

다음에 소개하는 소스 코드는 fileReader 객체를 사용해 데이터를 가져오는 VanillaDataReaderClient 클래스를 보여준다.

```
public class VanillaDataReaderClient {
  private VanillaFileReader fileReader = null;
  private String fileName
```

```
    = "src/main/resources/basics/basics-trades-data.txt";

  public VanillaDataReaderClient() {
    try {
      fileReader = new VanillaFileReader(fileName);
    } catch (FileNotFoundException e) {
      System.out.println("Exception" + e.getMessage());
    }
  }

  private String fetchData() {
    return fileReader.read();
  }

  public static void main(String[] args) {
    VanillaDataReaderClient dataReader = new VanillaDataReaderClient();
    System.out.println("Got data using no-spring: " +
                       dataReader.fetchData());
  }
}
```

이름에서 알 수 있듯이 VanillaDataReaderClient 클래스는 데이터 소스로부터 데이터를 가져오는 클라이언트다. 이를 실행할 때 VanillaDataReaderClient 클래스는 생성자 안에서 참조하는 fileReader 객체와 함께 인스턴스화한 후, fileReader 객체를 사용해서 결과 값을 가져온다.

다음은 VanillaFileReader 클래스를 구현한 것이다.

```
public class VanillaFileReader {
  private StringBuilder builder = null;
  private Scanner scanner = null;
```

```
    public VanillaFileReader(String fileName) throws FileNotFoundException {
      scanner = new Scanner(new File(fileName));
      builder = new StringBuilder();
    }

    public String read() {
      while (scanner.hasNext()) {
        builder.append(scanner.next());
        builder.append(",");
      }
      return builder.toString();
    }
  }
```

간단하고 쉽지 않은가?

방금 소개한 VanillaDataReaderClient 클래스는 오직 파일 시스템에서만 데이터를 읽어올 수 있다는 한계가 있다. 그런데 어느 날 아침, 여러분의 상사가 파일이 아니라 데이터베이스나 소켓에서 데이터를 읽어올 수 있는 프로그램을 급하게 요구했다고 생각하자. 하지만 위 소스 코드를 살펴보면 오직 fileReader 객체만 VanillaDataReaderClient 클래스에 연계되어 있기 때문에 파일이 아닌 자료는 접근할 수 없다. 때문에 현재 소스 코드로는 이러한 수정 사항을 추가할 수 없다. 상사의 요구 사항을 만족하는 프로그램으로 고치려면 DatabaseReader 클래스 객체나 SocketReader 클래스 객체를 VanillaDataReaderClient 클래스에서 사용할 수 있어야 한다.

끝으로(가장 중요한 부분이다) 방금 소개한 클래스 객체와 VanillaFileReader 클래스 객체가 매우 강하게 결합한 것이 보이는가? 이는 VanillaDataReaderClient 클래스가 VanillaFileReader 클래스의 변화에 민감하다는 말이다. 즉,

VanillaFileReader 클래스를 변경하면 VanillaDataReaderClient 클래스도 같이 변경해야 한다는 뜻이다. 이미 많은 사람이 VanillaDataReaderClient 클래스를 같이 사용하고 있는데, 이것을 대대적으로 수정해야 한다고 가정하자. 이 클래스를 리팩토링하는 것은 쉬운 일이 아니다. 추후에 수정이 용이하고 검증이 편리한 컴포넌트를 만들고자 한다면 객체 간의 강한 결합을 피하는 편이 좋다.

그럼 상사가 요구한, 어떤 데이터 소스든 읽어올 수 있는 프로그램을 만들어보자. 즉, 이 프로그램을 한 단계 더 발전시켜 어떤 형태의 데이터 소스에서든 읽어올 수 있도록 리팩토링해보자. 이를 위해서는 유명한 '인터페이스를 이용한 설계' 원칙을 살펴보자.

1.2.1 인터페이스를 이용해 설계하기

이 책에서 설명하는 설계 방법들이 가장 좋은 방법이라고 할 수는 없지만 인터페이스를 이용해 소스 코드를 작성하는 일은 매우 좋은 습관이다. 인터페이스를 설계하는 첫 번째 단계는 일단 인터페이스를 만드는 것이다.

인터페이스의 구현 클래스는 이 인터페이스의 명세에 맞게 구현하면 된다. 인터페이스 명세를 수정하지 않는 한, 클라이언트(인터페이스를 사용하는 클래스)는 구현 클래스의 내용 변화에 영향을 받지 않는다. 클라이언트를 손대지 않고도 구현 클래스를 몇 번이고 수정할 수 있다.

따라서 데이터를 읽는 프로그램을 구현하려고 메서드 하나를 가진 IReader 인터페이스를 다음처럼 작성했다.

```
public interface IReader {
  public String read();
}
```

다음에는 인터페이스의 규약 부분을 구현해보자. 여러 가지 데이터 소스에서 데이터를 읽어야 하므로 파일 시스템에서 읽어올 때 필요한 FileReader, 데이터베이스에서 읽어올 때 필요한 DatabaseReader, FTP 서버에서 읽어올 때 필요한 FtpReader 등 각각의 실제 구현 클래스를 생성한다.

XXXReader 형태로 구현하는 클래스 템플릿template은 다음과 같다.

```
public class XXXReader implements IReader {
  public String read() {
    // 여기에 구현
  }
}
```

XXXReader 클래스를 생성했다면 다음 단계는 이 클래스를 클라이언트 프로그램에서 사용하는 일이다. 하지만 여기서는 구현 클래스를 직접 참조하기보다는 인터페이스를 참조할 것이다.

다음 예처럼 FileReader 클래스나 FtpReader 클래스가 아니라 IReader 인터페이스 변수인 reader를 참조 값으로 가지도록 수정한 DataReaderClient 클래스를 살펴보자. 이 클래스는 reader 변수를 매개변수로 취하는 생성자를 가진다.

```
public class DataReaderClient {
  private IReader reader = null;
  private static String fileName
    = "src/main/resources/basics/basics-trades-data.txt";

  public DataReaderClient(IReader reader) {
    this.reader = reader;
  }
```

```
    private String fetchData() {
      return reader.read();
    }

    public static void main(String[] args) {
      ...
    }
  }
```

소스 코드를 살펴봤을 때 실제로 어떤 형태의 구현 클래스(XXXReader)를 사용했는지 물어본다면 누군가 대답할 수 있을까? 할 수 없다!

DataReaderClient 클래스는 실제 프로그램이 실행되기 전까지 어떤 데이터를 받을지 알 수 없다. IReader 인터페이스 타입인 reader가 어떤 구현 클래스를 사용하게 되는지는 런타임의 다형성에 의해 결정되기 때문이다. 여기에서 중요한 사실은 클라이언트가 IReader 인터페이스의 형태를 가지는 어떤 클래스라도 사용할 수 있다는 것이다. 즉, 각 IReader 인터페이스에 구현한 인터페이스 메서드는 적절한 시기에 호출할 수 있다.

IReader 인터페이스에서 구현한 구현 클래스(XXXReader)를 클라이언트에 적절하게 제공하는 일은 어렵다. 다음 소스 코드처럼 IReader 인터페이스를 인스턴스화하는 것(FileReader 클래스의 객체인 fileReader를 생성하고 fileName이라는 변수를 클라이언트에 매개변수로 전달한다)이 방법이다.

```
  public class DataReaderClient {
    private IReader reader = null;
    private static String fileName
      = "src/main/resources/basics/basics-trades-data.txt";

    ...
```

```
    public static void main(String[] args) {
      //음.. 여전히 강하게 결합되어 있나?
      IReader fileReader = new FileReader(fileName);
      DataReaderClient client = new DataReaderClient(fileReader);
      System.out.println("Got data using interface design priciple: "
          + client.fetchData());
    }
  }
```

하지만 여전히 IReader 인터페이스와 구현 클래스는 결합되어 있다고 할 수 있다. IReader 인터페이스에서 어떤 클래스를 사용할지 DataReaderClient 클래스에서 알려주어야 하기 때문이다.

물론 각 구현 클래스(XXXReader)는 IReader 인터페이스에 이미 구현했으므로, FileReader 클래스를 DatabaseReader나 FtpReader 클래스로 변경하는 일은 어렵지 않다. 즉, 일을 요청하는 관리자가 갑자기 요구 사항을 바꿨을 때 이에 대응하기가 훨씬 용이하다는 뜻이다.

하지만 IReader 인터페이스의 구현 클래스가 아직도 DataReaderClient 클래스에 결합해 있기 때문에 이 결합을 최대한으로 제거하는 것이 가장 이상적이다. 하지만 문제는 강하게 결합하지 않고도 어떻게 IReader 인터페이스의 인스턴스를 DataReaderClient 클래스에 제공할 수 있는가다.

FileReader 클래스를 생성했다는 것을 DataReaderClient 클래스에서 알 필요가 없도록 추상화할 수 있을까? 더 많은 질문을 하기 전에 대답부터 하자면 답은 '가능하다'다. 어떠한 의존성 주입 프레임워크를 사용해도 이를 해결할 수 있으며, 이런 프레임워크의 하나가 스프링이다.

스프링은 런타임에 객체들간의 의존성을 해결해주는 역할을 하는 의존성 주입(혹은 제어-역전Inversion of Control, IoC) 프레임워크다. 스프링을 이용해 해결하고자 하는 IReader 인터페이스 문제를 해결하기 위해서는 어떻게 해야 할까? 클래스들 간의 의존성을 파악하는 것이 우선이 될 것이다. 클래스들 간의 의존성을 파악하는 방법에 대해서는 이 책에서 더 이상 자세히 다루지는 않겠다.

1.3 스프링 소개

객체 상호작용은 소프트웨어 개발에서 중요한 부분이다. 좋은 설계는 이미 만들어진 소스 코드에 아무 영향을 끼치지 않거나 혹은 아주 미약한 영향만 끼치면서 동적인 부분의 변경을 가능하게 한다. 이렇게 동적인 부분끼리 매우 긴밀하게 연결되어 있을 때 객체가 강하게 결합했다고 말하는데, 이런 구조의 설계는 유연성이 없다. 왜냐하면 가변성이 없고 독립적인 상태에서는 검증할 수 없으며, 소스 코드를 어느 정도 변경하지 않고서는 유지보수가 불가능하기 때문이다. 스프링은 의존성을 없애고 컴포넌트를 설계하는 수고를 덜기 위해 만들었다.

1.3.1 의존성 주입

스프링은 의존성 주입이라는 하나의 목표에 집중한 프레임워크며, 이는 종종 제어-역전 원칙과 혼용해 사용되기도 한다. 보통의 독립형standalone 프로그램은 main 프로그램부터 시작하며, 의존성을 생성한 후 적절한 메서드를 실행하는 순서로 실행한다. 하지만 제어-역전은 정확히 반대의 순서로 실행한다. 즉, 모든 의존성과 관계성은 제어-역전 컨테이너가 만들고, 이들을 프로퍼티로 main 프로그램에 주입하면 프로그램은 동작 대기 상태가 된다. 이처럼 보통의 프로그램 생성 과정과 정반대이므로 제어-역전 원칙이라고 부른다.

1.3.2 스프링으로 IReader 인터페이스 리팩토링하기

IReader 프로그램을 다시 살펴보자. 클래스 사이의 강한 결합을 해결하는 방법은 IReader 인터페이스에서 구현한 구현 클래스를 DataReaderClient 클래스에 강제로 주입하는 것이다.

다음 소스 코드처럼 DataReaderClient 클래스를 결합되지 않는 상태로 수정하는 DecoupledDataReaderClient 클래스를 살펴보자.

```
public class DecoupledDataReaderClient {
  private IReader reader = null;
  private ApplicationContext ctx = null;

  public DecoupledDataReaderClient() {
    ctx = new ClassPathXmlApplicationContext("basics-reader-beans.xml");
  }

  private String fetchData() {
    reader = (IReader) ctx.getBean("reader");
    return reader.read();
  }

  public static void main(String[] args) {
    DecoupledDataReaderClient client = new DecoupledDataReaderClient();
    System.out.println(
      "Using Decoupled Data Client, Got data: " + client.fetchData()
    );
  }
}
```

보다시피 ApplicationContext 인터페이스 때문에 마법 같은 일들이 많이 일어난다! 무슨 일이 일어나는지 간단히 살펴보자.

JVM이 DecoupledDataReaderClient 클래스를 인스턴스화하면 클래스의 생성자 안에서 스프링의 ClassPathXmlApplicationContext 클래스 객체인 ctx를 생성한다. ApplicationContext 인터페이스는 XML 형식의 설정 파일을 읽어 파일에서 선언한 모든 객체를 생성한다. 이 예제에서는 reader라는 이름의 FileReader 클래스 객체를 생성한다.

그리고 ApplicationContext 인터페이스를 인스턴스화하면 XML 파일에서 정의한 객체를 포함하는 컨테이너를 생성한다. 그런 다음 fetchData 메서드를 호출하면 컨테이너로부터 reader라는 빈bean을 가져와 FileReader 클래스에서 선언한 적절한 메서드를 실행할 것이다. 이에 대해서는 자세히 알지 못하더라도 1장의 뒷부분에서 스프링의 기본을 설명할 것이므로 걱정할 필요는 없다. 지금은 예제를 계속 살펴보자.

DecoupledDataReaderClient 클래스를 생성한 후, 다음처럼 FileReader 클래스의 정의definition를 포함한 XML 파일을 작성하자.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean name="reader" class="com.madhusudhan.jscore.basics.FileReader">
    <constructor-arg value
      ="src/main/resources/basics/basics-trades-data.txt" />
  </bean>
</beans>
```

위 XML 파일은 각각의 빈bean(클래스의 인스턴스)을 생성하고 관계성을 결정하는 데 목적이 있다. 따라서 빈, 관계성과 함께하는 객체 그래프를 가진 컨테이너container를 생성하는 ApplicationContext 인터페이스를 인스턴스화할 때 사용할 수 있다.

앞 XML 파일의 reader 빈을 확인해보자. ctx = new ClasspathXmlApplicationContext("basic-reader-beans.xml") 구문은 basic-reader-beans.xml 파일에서 정의한 빈 컨테이너를 생성한다. 그리고 FileReader 클래스 객체는 스프링에서 reader라는 이름으로 생성하고 컨테이너에 저장한다.

즉, 스프링 컨테이너는 단순히 XML 파일에서 생성한 빈 인스턴스를 가진 보관소라고 볼 수 있다. 이러한 빈을 컨테이너에서 조회할 수 있는 API를 제공하므로 DecoupledDataReaderClient 클래스는 빈을 적절히 사용하면 된다.

예를 들어 ctx.getBean("reader")라는 API 메서드로 각 빈 인스턴스에 접근할 수 있다. 다음은 fetchData 메서드로 이 구문을 DecoupledDataReaderClient 클래스에 적용한 예다.

```
reader = (IReader) ctx.getBean("reader");
```

여기서 얻은 빈은 완전히 인스턴스화한 reader 빈이므로 이제 reader.read() 같은 메서드를 정상적으로 호출할 수 있다.

이제 스프링을 이용해 의존성 없는 프로그램이 동작할 수 있도록 하기 위해 무엇을 했는지 정리해보자.

- FileReader 클래스를 간단한 POJOPlain Old Java Object로 생성하는 IReader 인터페이스를 구현했다.

- 빈을 선언하는 간단한 XML 파일을 작성했다.
- 클라이언트에서 XML 파일을 읽어 빈을 로드하고 (ApplicationContext 인터페이스를 사용해) 이 빈들을 관리하는 컨테이너를 생성했다.
- 각 메서드를 호출할 수 있도록 완전한 형태의 FileReader 클래스 인스턴스를 얻으려고 컨테이너를 조회했다.

간단하고 명확하지 않은가?

혹시나 이 내용을 궁금해할지도 모를 독자를 위해 짧게 이야기해야 할 것이 있다. 보통 자바에서는 새로운 객체를 생성할 때 new 연산자를 사용한다. 그런데 위 DecoupledDataReaderClient 클래스에서는 reader 객체를 사용할 때 new 연산자를 사용하지 않았음을 눈치챘는가? 스프링은 이러한 역할을 개발자 대신 처리해준다. 즉, XML 파일에서 선언한 정의를 살펴보고 해당 객체들을 생성한다.

그리고 한 가지 더 살펴볼 것이 있다. DecoupledDataReaderClient 클래스와 XXXReader 클래스는 여전히 결합해 있다! 즉, 여전히 XML 파일에서 정의한 유형의 IReader 인터페이스 객체를 클라이언트로 주입해야만 한다. 사실 DecoupledDataReaderClient 클래스는 IReader 인터페이스를 전혀 모르는 것이 가장 이상적이다.

대신에 IReader 인터페이스와 DataReaderClient 클래스를 분리할 수 있는 서비스 계층을 생성할 수 있다. 이때 서비스 계층은 IReader 인터페이스보다는 DecoupledDataReaderClient 클래스에 연결할 수 있다.

그럼 서비스를 생성해보자.

1.3.3 ReaderService 클래스

ReaderService 클래스는 IReader 인터페이스의 세부 구현 사항을 클라이언트에서 분리하는 간단한 클래스다. DecoupledDataReaderClient 클래스는 해당 서비스만 알고 있으며, 서비스가 어디서 데이터를 얻어오는지는 전혀 알 수 없다.

첫 번째 단계는 서비스 클래스를 만드는 것이다. 다음 소스 코드는 서비스 클래스인 ReaderService를 보여준다.

```
public class ReaderService {
  private IReader reader = null;

  public ReaderService(IReader reader) {
    this.reader = reader;
  }

  public String fetchData() {
    return reader.read();
  }
}
```

ReaderService 클래스는 IReader 인터페이스 타입의 클래스 변수를 가지며, 생성자를 통해 IReader 인터페이스에서 구현한 클래스 객체를 주입받는다. 또한 데이터를 받으려고 각각 구현한 클래스 객체를 호출하는 역할을 하는 fetchData 메서드만 가진다. 다음 XML 파일은 ReaderService 클래스 객체를 basics-reader-beans.xml 파일의 적절한 IReader 인터페이스 객체와 연결한다.

```
<beans>
  <bean name="readerService"
        class="com.madhusudhan.jscore.basics.service.ReaderService">
    <constructor-arg ref="reader" />
```

```
    </bean>

    <bean name="reader"
          class="com.madhusudhan.jscore.basics.readers.FileReader">
      <constructor-arg
          value="src/main/resources/basics/basics-trades-data.txt" />
    </bean>
  </beans>
```

위 소스 코드를 자세히 살펴보면 ApplicationContext 인터페이스가 XML 파일을 읽는 시점에서 ReaderService 클래스 객체와 FileReader 클래스 객체를 인스턴스화한다. 그리고 ReaderService 클래스는 reader 빈을 참조(constructor-arg ref="reader")하므로, FileReader 클래스 객체를 먼저 인스턴스화하고, readerService 빈에 주입한다는 점을 주의하기 바란다.

reader 빈을 어떤 이유에서든 먼저 인스턴스화하지 못하면 스프링은 예외 상황을 발생시키고 프로그램을 바로 종료할 것이다. 즉, 스프링은 fail-fast 방식(순차적 접근에 실패하면 예외를 발생하는 방식)으로 설계되었다는 뜻이다.

다음은 ReaderService 클래스를 사용하게 한 ReaderServiceClient 클래스다.

```
public class ReaderServiceClient {
  private ApplicationContext ctx = null;
  private ReaderService service = null;

  public ReaderServiceClient() {
    ctx = new ClassPathXmlApplicationContext("basics-reader-beans.xml");
    service = (ReaderService) ctx.getBean("readerService");
  }

  private String fetchData() {
```

```
    return service.fetchData();
  }

  public static void main(String[] args) {
    ReaderServiceClient client = new ReaderServiceClient();
    System.out.println("Got data using ReaderService: "
        + client.fetchData());
  }
}
```

주목할 점은 ReaderServiceClient 클래스가 XXXReader 클래스 등은 전혀 모르고 오직 ReaderService 클래스만 안다는 점이다. 게다가 컴포넌트의 결합을 풀어내기까지 했다.

데이터베이스에서만 데이터를 읽는다면 XML 파일을 제외하고는 소스 코드를 전혀 변경할 필요가 없다는 점을 기억하자. 즉, XML 파일 안의 적절한 XXXReader 클래스 객체를 동작시키고 프로그램을 다시 실행하면 된다. 예를 들어 다음 소스 코드를 살펴보면, 기존 소스 코드는 한 줄도 고치지 않고 메타데이터metadata만 수정해 reader 빈이 DatabaseReader 클래스 객체를 생성하도록 변경했다.

```
<bean name="readerService"
      class="com.madhusudhan.jscore.basics.service..ReaderService">
  <!-- 현재 DBReader를 나타내는 리더 -->
  <property name="reader" ref="reader" />
</bean>

<!-- *** DBReader ** -->
<bean name="reader"
      class="com.madhusudhan.jscore.basics.readers.DatabaseReader">
  <property name="dataSource" ref="dataSource" />
```

```
</bean>

<!-- DBReader에 의존하는 데이터 리소스 -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
  ....
</bean>
```

이제 프로그램을 실행할 때 ReaderService 클래스는 동적으로 적절한 IReader 인터페이스를 참조할 것이다.

1.4 요약

1장에서는 스프링을 매우 개괄적으로 소개했다. 객체의 결합도 문제와 스프링으로 어떻게 의존성 문제를 해결할 수 있는지 배웠으며 컨테이너도 살펴봤다. 이제 스프링 사용 방법을 간략하게 살펴봤으니, 아직 남아있는 기초적인 내용을 다음 장에서 다룰 것이다. 2장에서는 스프링의 내부를 좀 더 깊게 알아보도록 하자.

2 | 스프링 빈

1장에서는 스프링의 가장 기초적인 부분을 살펴보면서 빈, 빈 팩토리bean factory, 컨테이너 등을 배웠다. 2장에서는 빈을 어떻게 만드는지, 어떻게 이름을 짓는지, 어떻게 컨테이너와 연결하는지 등등 1장에서 배운 내용을 조금 더 깊게 살펴본다.

2.1 빈 환경 설정

스프링에서는 모든 객체가 빈이다! 즉, 객체를 빈으로 정의하는 일이 스프링의 가장 핵심이다. 본래 빈이란 스프링이 클래스 정의definition를 조회해서 생성하고 관리하는 객체 인스턴스 그 이상도 이하도 아니다. 이 정의란 기본적으로 XML 파일의 메타데이터에서 가져온다. 그러면 스프링은 이 메타데이터에 기반을 두고 어떤 객체를 인스턴스화해야 하는지, 어떤 의존성을 결정하고 주입해야 하는지, 새롭게 생성한 인스턴스의 범위는 무엇인지 등에 대한 계획을 짠다.

메타데이터는 1장에서 살펴봤던 파일과 비슷하게 간단한 XML 파일로 받을 수 있다. 아니면 어노테이션annotation이나 자바 설정 파일 형태로도 메타데이터를 받을 수 있다. 먼저 XML을 사용해 스프링 빈의 정의를 확인하는 방법을 알아보자.

2.1.1 XML을 이용한 설정

스프링은 XML 파일에서 정의한 자바 클래스를 읽어 이를 초기화하고 런타임 컨테이너에 스프링 빈으로 로드한다는 점은 이미 설명한 바 있다. 여기서 컨테이너란 런타임 시 완벽하게 준비한 자바 클래스의 인스턴스 모두를 담는 보관소다. 이제 프로세스가 어떻게 동작하는지 예제를 통해 알아보자.

Person 클래스 객체에 대응하는 person이라는 빈을 하나 정의할 것이다. 이 빈은 세 개의 프로퍼티를 가지며, 그중 두 개(firstName과 lastName)는 생성자를 통해 지정할 것이고 세 번째 프로퍼티(age)는 setter 메서드를 사용해 지정할 것이다. 여기에 추가로 address 프로퍼티도 존재하는데, 이 프로퍼티는 자바 클래스가 아니라 다른 클래스의 주소를 가리키는 참조 값이다.

다음 소스 코드는 빈을 위한 Person이라는 클래스를 구현한다.

```
public class Person {
  private int age = 0;
  private String firstName = null;
  private String lastName = null;
  private Address address = null;

  public Person(String fName, String lName) {
    firstName = fName;
    lastName = lName;
  }

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  public Address getAddress() {
    return address;
  }
```

```
  public void setAddress(Address address) {
    this.address = address;
  }

  public String getDetails(){
    return firstName + " " + lastName + " is " + getAge()
           + " old and lives at " + getAddress();
  }
}
```

보다시피 age와 address 변수만 setter 메서드를 가진다. 즉, 생성자로 만드는 변수와는 다른 방식으로 값을 설정한다.

Address 클래스는 다음처럼 구현한다.

```
public class Address {
  private int doorNumber = 0;
  private String firstLine = null;
  private String secondLine = null;
  private String zipCode = null;
  // 해당 변수의 getter와 setter 메서드는 여기에 구현
  ....
}
```

최종적으로 해야 할 일은 XML 파일에 person과 address 빈을 생성하는 것이다. 해당 클래스는 다음처럼 XML 파일에 선언해야 한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="person" class="com.madhusudhan.jscore.beans.Person">
      <constructor-arg value="Madhusudhan" />
      <constructor-arg value="Konda" />
      <property name="age" value="99" />
      <property name="address" ref="address" />
    </bean>

    <bean name="address" class="com.madhusudhan.jscore.beans.Address">
      <property name="doorNumber" value="99" />
      <property name="firstLine" value="Rainbow Vistas" />
      <property name="secondLine" value="Kukatpally, Hyderabad" />
      <property name="zipCode" value="101010" />
    </bean>
  </beans>
```

이때 주의할 몇 가지 사항이 있다. 여기서 최상위 노드가 〈beans /〉 요소를 루트 요소로 선언했으므로 모든 빈 정의에는 〈bean /〉 요소가 있어야 한다. 보통 XML 파일은 적어도 빈 하나를 가지며 각 빈에는 가장 중요한 정보인 name과 class 속성을 가진다. 또한 id 속성, scope 속성, 의존성과 관련한 속성과 같은 추가 정보를 가졌을 수도 있다.

기본적으로는 런타임 시 XML 파일을 로드할 때 스프링에서 이 정의들을 읽은 후 Person 클래스의 인스턴스를 만들고, name 속성에 person이라는 값을 부여한다. 그러면 스프링 API를 사용해 컨테이너에서 해당 인스턴스를 조회할 때 이 이름을 사용할 수 있다.

다음은 PersonClient 클래스가 어떻게 컨테이너를 로드해 빈을 조회할 수 있는지를 보여주는 예제다.

```
public class PersonClient {
  private static ApplicationContext context = null;
  public PersonClient() {
    context = new ClassPathXmlApplicationContext("ch2-spring-beans.xml");
  }
  public String getPersonDetails() {
    Person person = (Person) context.getBean("person");
    return person.getDetails();
  }
}
```

위 소스 코드는 다음처럼 중요한 두 단계를 거친다.

- ApplicationContext 인터페이스는 빈을 정의한 XML 파일에 의해 인스턴스화를 한다. 이때 context 변수는 그저 빈의 보관소일 뿐이며, 스프링에서는 컨테이너가 된다.
- 컨테이너에서는 새로 생성한 person 빈을 조회한다. 자바 인스턴스를 검색하는 데는 스프링 컨텍스트 API의 getBean 메서드를 사용한다. getBean 메서드로 조회해서 전달한 문자 값은 XML 파일에 있는 빈의 이름이며, 이 예제에서는 person이다.

여러 XML 파일로 나눠 빈을 정의할 수도 있다. 예를 들면 업무 기능과 관련된 빈은 business-beans.xml에, 유틸리티와 관련된 빈은 util-beans.xml에, 데이터 접근과 관련된 빈은 dao-beans.xml에 만드는 방법 등이다. 어떻게 여러 XML 파일을 스프링 컨테이너로 인스턴스화하는지는 2장의 뒷부분에서 알아볼 것이다.

일반적으로 XML 파일을 작성할 때 파일 이름은 두 부분을 하이픈으로 나눈 방식을 사용하는 편이다. 첫 번째 부분은 기능적인 면을 대변하고 두 번째 부분은 단순

하게 이것이 빈임을 나타낸다. 단, 파일 이름을 짓는 방법에 제한은 없으니 원하는 대로 빈 이름을 만들어 쓰면 된다.

빈 각각은 name 속성이나 id 속성을 반드시 가져야 한다. 물론 name이나 id 속성 둘 다 없는 익명의 빈을 만들 수는 있지만 해당 빈을 클라이언트 소스 코드에서 조회할 수가 없다. id와 name 속성은 빈의 id 속성 값이 XML 스펙에 명시한 특별한 값일 때를 제외하고는 같은 용도로 쓴다. 이 말은 빈을 조회할 때 id 속성 값을 확인한다는 의미(예를 들면 id 속성 값에 특수 문자를 쓰면 안 된다는 등)다. name 속성에는 아무 제한 사항이 없다.

class 속성에는 클래스 이름을 선언한다. 클래스를 인스턴스화할 때 초기화한 데이터가 필요하면 이는 프로퍼티나 생성자의 인자로 값을 지정한다. 예를 들면 Person 클래스는 생성자 인자, setter 프로퍼티 모두를 사용해 인스턴스화했다. 따라서 20쪽 XML 파일에서 firstName과 lastName는 〈constructor-arg /〉 요소의 value 속성을 사용해 값을 지정했고, 그 외는 단순한 〈property /〉 요소의 name 속성과 value 속성을 사용했다.

또한 해당 〈property /〉 요소의 value 속성이 가진 값은 그냥 단순한 값일 수도 있고, 다른 빈의 참조 값일 수도 있다. address 프로퍼티처럼 어떤 빈을 다른 빈과 연결할 필요가 있다면 ref 속성을 사용하면 된다. 〈property name="address" ref="address" /〉처럼 다른 빈을 참조해야 할 때는 value 속성보다 ref 속성을 사용함을 잊지 말자.

빈 이름은 원하는 대로 지을 수 있지만 첫 번째 글자를 소문자로 하는 카멜 표기법 CamalCase 사용을 권한다.

2.1.2 어노테이션 이용

어노테이션Annotation을 이용해 빈을 연결할 수도 있다. 즉, 빈을 정의하고 연결할 때 어노테이션을 이용하면 효과적으로 XML 메타데이터를 간소화할 수 있다.

이제 어노테이션으로 어떻게 빈을 정의하는지 알아보자. ReservationManager 클래스는 ReservationService 클래스에 의존성을 가지기 때문에 비행기 예약 작업을 위해서는 서비스 프로퍼티로 사용하려는 ReservationService 클래스를 예약 관리를 위한 ReservationManager 클래스에 주입해야만 한다.

다음은 비어있는 ReservationService 클래스를 선언한다.

```
public class ReservationService {
  public void doReserve(ReservationMessage msg) {
    //
  }
}
```

ResevationManager 클래스는 비행기 예약을 처리하는 ReservationService 클래스에 의존성을 가진다. 이러한 의존성을 만족하려면 서비스에 해당하는 ReservationService 클래스를 ResevationManager 클래스에 주입해야만 한다.

이번에는 XML 파일을 이용했을 때와는 다르게 @Autowired라는 어노테이션을 사용해 의존성을 주입한다. 다음 소스 코드에서 reservationService라는 변수에 이 어노테이션을 어떻게 사용했는지 살펴보자.

```
public class ReservationManager {
  @Autowired
  private ReservationService reservationService = null;

  public void process(Reservation r) {
```

```
        reservationService.reserve(r);
    }
}
```

변수, 메서드, 생성자를 @Autowired로 어노테이션했다면 스프링은 관련 의존성을 찾아 자동으로 주입한다. 위 예제에서 ReservationManager 클래스는 ReservationService 클래스의 인스턴스를 찾는다. 이때 보이지 않는 곳에서 스프링은 byType 방식을 사용해 빈을 연결할 것이다(더 자세한 사항은 '3.5 오토와이어링'을 읽어보도록 하자).

마지막으로, 어노테이션을 사용할 것임을 스프링에 알려야 한다. 이를 위해 XML 파일에 특별한 요소를 선언한다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context.xsd">

  <context:annotation-config />
  <!-- 예약 서비스 부분 -->
  <bean name="resSvcABC" class
    ="com.madhusudhan.jscore.fundamentals.annotations.ReservationService" />
  <bean name="reservationManager" class
    ="com.madhusudhan.jscore.fundamentals.annotations.ReservationManager" />
</beans>
```

앞 소스 코드의 〈context:annotation-config /〉 요소는 스프링에게 빈이 어노테이션을 사용할 것임을 알 수 있게 해준다. 이렇게 하면 스프링은 이에 해당하는 어노테이션을 사용하는 클래스가 있는지를 찾아 해야 할 작업을 처리한다(이 예제에서는 @Autowired로 어노테이션한 ReservationManager 클래스가 존재한다). 이때 위 예제에서 진하게 표시한 부분처럼 적절한 context 스키마를 추가해야 함을 잊지 말자.

reservationManager 빈을 생성하면 스프링은 ReservationService 클래스 타입의 빈을 찾아 이를 주입한다(XML 파일에서 해당 빈을 생성해둔 바 있다). 여기서 ReservationService 클래스에 해당하는 빈의 name 속성 값을 resSvsABC로 지정했으므로 의존성을 확인할 때 스프링이 byName 방식보다는 byType 방식을 사용할 것이란 걸 알 수 있다. 따라서 ReservationManager 클래스에 해당하는 빈의 name 속성 값으로 reservationManager를 지정하면 resSvsABC 빈을 항상 주입할 것이다.

여기서 두 가지 사항을 좀 더 알아보자. reservationManager 빈에는 어떠한 프로퍼티도 선언한 적이 없다. 예를 들어 〈property name="reservationService" ref="resSvs" /〉 구문을 선언하지 않았으며, setter와 getter 프로퍼티도 존재하지 않는다. 이는 어노테이션 설정으로 깔끔하게 처리할 수 있으므로 필요 없는 작업이 되었다.

이 부분의 설명을 끝내기 전에 반드시 살펴봐야 할 속성이 있다. 바로 〈context /〉 요소의 네임스페이스를 설정하는 〈context:component-scan /〉 요소다. 위 예제에서는 XML 메타데이터의 많은 부분을 간소화하는 일이 가능했지만, 여전히 서비스 빈은 선언해야만 한다.

그럼 이런 번거로움을 피하고 XML 메타데이터를 더 간소화할 방법은 없을까? 답을 먼저 말하자면 '있다'다. 방금 설명한 〈context:component-scan /〉 요소를

사용하면 XML 메타데이터 대부분을 없앨 수 있다. 요소 이름에서 알 수 있듯이 특별하게 어노테이션한 클래스를 찾기 위해 특정 디렉터리를 검색하는 역할을 한다.

그럼 ReservationService 클래스를 @Component를 사용해 어노테이션해보자.

```
@Component
public class ReservationService {
    ..
}
```

스프링은 @Component 어노테이션을 사용한 모든 클래스를 찾아내어 인스턴스화한다(해당 클래스가 〈context:component-scan /〉 요소의 base-package 속성 값일 경우다). 따라서 다음에 해야 할 일은 〈context:component-scan /〉 요소를 다음처럼 선언하는 것이다.

```
〈context:component-scan
  base-package="com.madhusudhan.jscore.fundamentals.annotations" /〉
```

위 소스 코드와 같이 〈context:component-scan /〉 요소를 선언하면 스프링에게 base-package 속성에 지정한 패키지 중, @Component로 어노테이션한 클래스를 찾으라고 알려준다는 뜻이다. 이에 따라 스프링은 reservationService 빈을 찾아 해당 클래스를 인스턴스화할 것이며 @Autowired 어노테이션 때문에 reservationManager 빈으로 자동 주입한다.

@Autowired 어노테이션에 의존하고 싶지 않은 독자라면 JSR-330[01]의 @Inject 어노테이션을 사용하는 일도 고려하는 것이 좋다. 스프링은 이 어노테이션도 지원

01 스프링과 구글 Juice가 함께하는 의존성 주입 표준화 제안이다.

하지만 여기서는 @Inject 어노테이션을 다루지는 않는다. @Autowired와 거의 유사하다고 알아두자. 참고로 XML 파일이 아닌 어노테이션을 사용하는 일에는 많은 논쟁이 있다는 것도 알아두자.

어노테이션을 사용하면 XML 파일을 깔끔하게 정리할 수 있지만 소스 코드와 밀접하게 묶이게 된다. 그런 이유로 학계에는 어노테이션의 사용을 찬성하는 사람이 있는가 하면 반대하는 사람도 있다. 개인적으로는 설정과 관련한 문제를 해결하는 깔끔하고 간편한 접근 방법 때문에 어노테이션을 좋아하지만 메타데이터만은 못하다. 메타데이터는 애플리케이션을 다시 컴파일하거나 빌드할 필요 없이 설정을 수정할 수 있기 때문이다.

2.1.3 XML 네임스페이스

가끔 XML 파일이 지나치게 많은 빈 정보를 가질 때가 있는데, 이는 보기에 좋지 않다. 그렇지만 XML 파일을 기반으로 둔 스키마를 사용하면 이를 효과적으로 간소화할 수 있다.

지금까지 다음 소스 코드가 보여주는 것처럼 메타데이터로 작성한 XML 스키마를 사용해왔다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
  <!-- 여기에 빈을 선언 -->
  <bean>
    ...
  </bean>
</beans>
```

보다시피 앞 XML 파일에서는 〈beans /〉 요소에 스키마를 사용했다. 이외에도 스프링은 jms, aop, jee, tx, lang, util 등과 같은 다양한 스키마를 정의해놓았다. 적절한 스키마를 찾아 추가하는 방법은 굉장히 쉽다. 예를 들어 jms 스키마를 사용하고 싶다면 설정 파일에 다음 부분(볼드체)을 추가하면 된다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jms="http://www.springframework.org/schema/jms"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
         http://www.springframework.org/schema/jms
         http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">
       ....
</beans>
```

xmlns 속성은 jms 스키마를 설정한 네임스페이스를 정의하며 마지막 두 줄은 스키마 정의가 위치한 곳을 가리킨다. 즉, 아래 패턴의 xxx 부분에 사용하고 싶은 스키마를 넣어서 사용하면 된다.

```
<!-- xxx 부분을 jms, aop, tx 등 사용하고 싶은 스키마를 선택해 사용한다 -->
http://www.springframework.org/schema/xxx
http://www.springframework.org/schema/xxx/spring-xxx-3.0.xsd
```

적절한 스키마와 관련 있는 네임스페이스(xmlns 속성을 사용한)를 알고 있다면 스키마를 이용한 설정은 식은 죽 먹기다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jms="http://www.springframework.org/schema/jms"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
          http://www.springframework.org/schema/jms
          http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">
    <jms:listener-container>
      ...
    </jms:listener-contaner>
  </beans>
```

그럼 jms 스키마를 설정한 네임스페이스를 사용해 listener-container 빈을 어떻게 컨테이너에 추가하는지 살펴보자.

예전의 DTD 스타일 설정도 여전히 사용할 수 있다.

```
  <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
            "http://www.springframework.org/dtd/spring-beans.dtd">
  <beans>
    ....
  </beans>
```

하지만 굳이 DTD 스타일을 사용할 이유가 없다면, 스키마를 사용한 설정을 추천하는 바다.

2.2 빈 생성하기

빈이란 애플리케이션의 이용 목적을 달성하기 위해 서로 묶은 인스턴스라고 볼 수 있다. 보통 일반 자바 애플리케이션에는 의존성이나 연관성을 포함한 컴포넌트의 라이프 사이클이 존재한다.

예를 들어 메인 클래스를 실행할 때 메인 클래스는 애플리케이션을 실행하는 데 필요한 모든 의존성을 결정하고, 프로퍼티를 지정하며, 의존하는 인스턴스를 인스턴스화한다.

독립형 애플리케이션에서는 의존성을 생성하거나 연결하고 이를 적절한 인스턴스에 연관시켜야 하는 권한이 모두 메인 클래스에 있다. 하지만 스프링에서는 모든 권한이 스프링에 있다. 이는 인스턴스(빈)를 만들고, 연관성을 성립시키며, 의존성을 주입하는 일을 전적으로 스프링이 처리한다는 뜻이다. 사실 여러분은 이러한 사항을 정의하고 로드하는 것 이외에 할 수 있는 일이 없다. 그러므로 모든 객체 그래프의 생성은 스프링의 책임이라고 볼 수 있다.

빈은 애플리케이션이 조회한 빈에 따라 행동할 수 있도록 컨테이너에 속한다. 물론 이를 위해서 빈의 연관성과 다른 메타데이터를 XML 파일에 선언하거나 스프링이 무엇을 해야 하는지 이해할 수 있도록 어노테이션 형태로 전달해야만 한다.

스프링의 중요한 특징 하나는 빈을 생성하는 과정에서 fail-fast 접근법을 따른다는 점이다. 이 말은 스프링이 빈을 로드할 때 무슨 문제가 생기면 바로 종료시킨다는 뜻이다. 즉, 전부가 아니라 반쯤 만들어진 빈을 가진 컨테이너란 존재하지 않는다.

이 부분은 런타임할 때가 아니라 컴파일할 때 모든 에러를 찾게 된다는 점에서 상당히 유용한 스프링의 특징이다.

2.3 라이프 사이클

스프링은 보이지 않는 곳에서 꽤 많은 일을 한다. 예를 들어 빈의 라이프 사이클은 이해하기 쉽지만 일반적인 자바 애플리케이션에서 사용한 라이프 사이클과는 다른 부분이 있다.

즉, 일반 자바에서 빈을 보통 new 연산자로 인스턴스화한다면 스프링에서는 단순히 빈을 생성하는 것 이외에 더 많은 일을 한다. 일단 빈을 생성하면 클라이언트에 해당하는 클래스는 빈에 접근할 수 있도록 적당한 컨테이너에 빈을 로드한다(컨테이너는 3장에서 배운다).

스프링에서 빈의 라이프 사이클은 다음과 같다.

- 스프링의 팩토리는 빈 정의를 로드하고 빈을 생성한다.
- 빈은 빈 정의에 선언한 프로퍼티대로 채워진다. 프로퍼티가 다른 빈을 참조하는 값이라면 다른 빈도 생성하고 입력하며, 참조성을 빈으로 주입한다.
- BeanNameAware나 BeanFactoryAware 인터페이스처럼 빈이 스프링의 인터페이스를 구현한다면 적절한 메서드를 호출한다.
- 스프링 프레임워크는 빈의 초기화 전 작업을 위해 빈 후처리기의 관련 메서드를 호출한다.
- 빈의 프로퍼티에 명시했다면 init-method 속성을 호출한다.
- 빈의 프로퍼티에 명시했다면 초기화 이후 작업을 진행한다.

위에 설명한 것들을 전부 이해할 수 없더라도 너무 스트레스를 받지는 말자. 좀 더 자세한 설명은 뒷부분에서 할 것이다.

일반적인 빈은 의존성을 갖지 않은 채 생성되는 반면 의존성을 가진 빈(즉, 다른 빈을 참조하는 프로퍼티를 가진)은 오직 빈이 가져야 할 의존성을 만족한 후에야 생성된다. 다음 예제를 통해 이 부분을 설명하겠다.

또한 정적 메서드와 팩토리를 사용해서 빈을 생성하는 일도 가능함을 기억하자. 이에 대해서는 3장에서 자세히 설명할 것이다.

2.3.1 의존성 없는 빈을 인스턴스화하기

1장에서 FileReader 클래스를 만들었음을 기억하는가? 기억하지 못한다면 소스 코드를 다시 살펴보자.

```
public class FileReader implements IReader {
  private StringBuilder builder = null;
  private Scanner scanner = null;

  // 생성자
  public FileReader(String fileName) { ... }

  // read 메서드 구현
  public String read() { ... }
}
```

FileReader 클래스의 생성자는 작업을 위해 fileName 변수를 인자로 얻는다. 그런 다음 메타데이터를 사용해 인자를 생성자에게 넘김으로써 빈을 정의할 수 있다. 다음의 빈 메타데이터를 확인해보자.

```
<bean name="reader"
      class="com.madhusudhan.jscore.basics.readers.FileReader">
  <constructor-arg
      value="src/main/resources/basics/basics-trades-data.txt" />
</bean>
```

필요한 fileName 변수는 src/main/resources/basics/basics-trades-data.txt 경로를 〈constructor-arg /〉 요소의 value 속성을 통해 인자로 전달하는 값으로 지정한다. 팩토리는 이 값을 읽고 new 연산자를 이용해 클래스의 객체를 생성한다(사실은 자바 리플렉션을 사용해서 빈을 인스턴스화하는 과정이다).

이제 이 빈은 의존성이 없으니 인스턴스화해 사용할 준비가 된 것이다.

2.3.2 의존성 있는 빈을 인스턴스화하기

이는 다른 빈에 의존하는 빈이 있을 때에 해당한다. 예를 들어 address 빈에 의존하는 person 빈을 생각해보자. address 빈으로 객체를 생성하지 않으면 person 빈은 객체를 생성하는 일이 불가능하다. 그러므로 person 빈은 address 빈에 의존한다고 말한다.

빈은 한 개 이상의 빈에 의존할 수 있다. 따라서 reader 빈이 또 다른 빈에 의존한다면 다른 빈을 생성하고 인스턴스화할 것이다. 다음 코드에서는 address라는 빈을 참조하는 address 프로퍼티를 가진 person 빈의 정의를 살펴볼 수 있다. 의존성을 만족시키려면 address 빈에서 생성한 객체를 person 빈에 주입해야 객체를 생성할 수 있다.

```
<bean name="person" class="com.madhusudhan.jscore.beans.Person">
  ....
  <property name="address" ref="address" />
</bean>

<bean name="address" class="com.madhusudhan.jscore.beans.Address">
  ....
</bean>
```

빈을 생성하는 순서는 스프링에서 매우 중요하다. 메타데이터를 읽은 후, 스프링은 의존성을 만족시키려고 생성해야 할 빈의 순서를 정하는 계획을 짠다(즉, 각 빈의 우선순위를 정한다). 그런 이유로 address 빈으로 객체를 생성하는 일을 person 빈으로 객체를 생성하기 전에 하는 것이다. 스프링이 address 빈으로 객체를 생성하는 과정에서 어떤 예외 상황이 발생했다면 프로그램은 바로 fail 되어 종료될 것

이다. 스프링은 이 단계에서 더는 빈을 만들지 않고 개발자에게 왜 진행하지 못했는지를 알려준다.

2.4 빈 별칭

가끔 같은 빈에 다른 이름을 지정해야 할 때가 있는데, 이때는 주로 별칭Alias을 사용한다. 예를 들어 address 빈은 shipping address나 billing address라고 부르는 경우도 있다. 즉, 별칭은 같은 빈을 다른 이름으로 만드는 방법이다. 이미 지정한 빈에 이름을 만들려면 alias 요소를 사용한다.

```
<bean name="address" class="com.madhusudhan.jscore.fundamentals.Address">
  ...
</bean>
<alias name="address" alias="billingAddress" />
<alias name="address" alias="shippingAddress" />
```

위 코드에서 address 빈은 원래 이름인 address로 선언했다. 그리고 두 개의 별칭 billingAddress와 shippingAddress를 선언해 모두 같은 빈을 가리키게 한다. 이제부터 이 별칭을 사용해 마치 원래 빈인 것처럼 사용할 수 있다.

2.5 익명의 빈

참조하는 빈하고만 관련되어 존재하는 빈도 만들 수 있다. 이러한 빈을 익명 혹은 내부 빈이라 부르는데, 이 빈은 이름이 없으므로 프로그램에서 빈을 확인하는 일은 불가능하다.

```
<bean name="reader"
      class="com.madhusudhan.jscore.basics.readers.FileReader">
  <constructor-arg
```

```
        value="src/main/resources/basics/basics-trades-data.txt" />
    <property name="platformLineEnder"
      <bean class="com.madhusudhan.jscore.basics.readers.WindowsLineEnder" />
    </property>
  </bean>
```

platformLineEnder 프로퍼티는 WindowsLineEnder 클래스의 빈을 참조하지만 platformLineEnder 빈을 프로퍼티로 정의했으므로(ref 속성이 붙지 않았음을 확인하자) reader 빈 이외의 다른 빈에는 사용할 수가 없다.

2.6 주입 방식

스프링에서 프로퍼티를 주입할 때는 생성자 그리고 setter 메서드를 이용하는 두 가지 방법이 있다. 둘 다 간단한 방식이므로 무엇을 사용하던 상관없다. 생성자를 사용할 때 생기는 단 하나의 이점은 추가로 setter 메서드의 소스 코드를 작성할 필요가 없다는 것이다. 그렇긴 해도 너무 많은 프로퍼티를 가진 생성자를 만드는 일은 좋은 방법이 아니다. 사실 개인적으로는 두 개 이상의 인자를 가진 생성자가 있는 클래스를 좋아하지 않는다.

2.6.1 생성자를 이용한 주입

이전 예제에서 〈constructor-arg /〉 요소를 사용해 생성자로 빈에 프로퍼티를 주입하는 생성자 주입 메서드를 설명한 바 있다. 기본적으로 클래스는 인자를 얻는 생성자를 가지며, 해당 인자는 XML 파일을 통해 연결한다.

다음 두 개의 인자를 얻는 생성자를 포함한 FtpReader 클래스의 소스 코드를 살펴보도록 하자.

```
public class FtpReader implements IReader {
  private String ftpHost = null;
```

```
  private int ftpPort = 0;

  public FtpReader(String ftpHost, int ftpPort) {
    this.ftpHost = ftpHost;
    this.ftpPort = ftpPort;
  }
  @Override
  public String read() {
    // 여기에 구현
    return null;
  }
}
```

ftpHost와 ftpPort 변수라는 인자는 XML 파일에 정의한 〈constructor-arg /〉 요소를 이용해서 연결했다.

```
<bean name="reader"
      class="com.madhusudhan.jscore.basics.readers.FtpReader">
  <constructor-arg value="madhusudhan.com" />
  <constructor-arg value="10009" />
</bean>
```

그 외 다른 빈에 생성자 인자를 참조 값으로 설정할 수 있다. 다음 소스 코드는 생성자를 이용해 reader 빈의 참조 값을 readerService 빈에 주입하는 예다.

```
<bean name="readerService"
      class="com.madhusudhan.jscore.basics.service.ReaderService">
  <constructor-arg ref="reader" />
</bean>

<bean name="reader"
```

```
        class="com.madhusudhan.jscore.basics.readers.FileReader">
    <constructor-arg
        value="src/main/resources/basics/basics-trades-data.txt" />
</bean>
```

다음은 ReaderService 클래스에 IReader 인터페이스 타입의 reader 인자를 사용해 생성자를 다루는 예다.

```
public class ReaderService {
  private IReader reader = null;

  public ReaderService(IReader reader) {
    this.reader = reader;
  }
  ...
}
```

인자 타입 결정

인자의 타입을 결정할 때는 스프링이 따라야 할 법칙이 있으므로 생성자를 이용한 주입에서 잠깐 짚고 넘어가겠다.

35쪽 FtpReader 예제를 살펴보면 생성자의 첫 번째 인자는 ftpHost고, 그 뒤로 ftpPort 인자가 따라온다. 이런 경우는 매우 명확하다. 생성자는 string과 int 타입의 값을 예상할 것이고, 스프링은 첫 번째 인자를 string 타입으로, 두 번째를 int 타입으로 선택하기만 하면 될 뿐이다. 비록 XML 파일 모두를 string 타입으로 선언했더라도 스프링에서 이용하는 java.beans.PropertyEditor가 string 값을 적절한 타입으로 변경하면 된다.

가장 명확한 방법은 다음 소스 코드에서 살펴볼 수 있듯이 인자의 타입을 선언문에 정의해놓는 것이다.

```
<bean name="reader"
      class="com.madhusudhan.jscore.basics.readers.FtpReader">
  <constructor-arg type="String" value="madhusudhan.com" />
  <constructor-arg type="int" value="10009" />
</bean>
```

인자의 타입은 보통 int, boolean, double, string 등과 같은 평범한 자바 타입을 사용하거나 다음처럼 0부터 시작하는 index 속성을 사용하는 경우도 있다.

```
<bean name="reader"
      class="com.madhusudhan.jscore.basics.readers.FtpReader">
  <constructor-arg index="0" type="String" value="madhusudhan.com" />
  <constructor-arg index="1" type="int" value="10009" />
</bean>
```

2.6.2 setter 메서드를 이용한 주입

의존성이 있는 빈과 프로퍼티를 생성자로 주입하는 방법 외에 스프링은 setter 메서드를 이용하는 방법도 제공한다. setter 메서드를 이용한 주입에서는 각 변수에 setter 메서드를 제공해야만 한다. 즉, 프로퍼티가 읽기/쓰기 특성을 모두 가진다면 변수에 setter 메서드와 getter 메서드를 제공하면 된다.

이제 ReaderService 클래스에는 IReader 인터페이스 타입의 변수를 생성하고 해당 프로퍼티에 대응하는 setter 메서드와 getter 메서드를 생성하면 된다. 참고로, setter 메서드를 이용해 프로퍼티를 전달했으므로 생성자는 비어있다(setter 메서드와 getter 메서드를 생성할 때는 일반적인 빈 생성 방법을 따르면 된다).

다음은 ReaderService 클래스를 수정한 소스 코드다.

```
public class ReaderService {
  private IReader reader = null;

  // setter 메서드와 getter 메서드
  public void setReader(IReader reader) {
    this.reader = reader;
  }

  public IReader getReader() {
    return reader;
  }
  ...
}
```

이때 IReader 인터페이스 타입의 reader를 인자로 갖는 setter와 getter 메서드, 생성자를 생략했다는 사실을 모두 기억하기 바란다. 메타데이터는 다음과 같다.

```
<bean name="readerService"
      class="com.madhusudhan.jscore.basics.service.ReaderService">
  <!-- setter 메서드의 인자를 주입 -->
  <property name="reader" ref="reader" />
</bean>

<bean name="reader"
      class="com.madhusudhan.jscore.basics.readers.FileReader">
  ...
</bean>
```

여기서는 reader라는 프로퍼티를 생성하고 reader 빈을 참조하게 했다. 그러면 스프링은 reader 프로퍼티를 위해 readerService 빈을 확인할 것이고, FileReader 클래스의 인스턴스를 전달하면서 setReader 메서드를 호출하게 될 것이다.

2.6.3 생성자와 setter 메서드의 혼용

원한다면 주입 방식을 짜맞추어 혼용하는 일도 가능하다.

몇 개의 프로퍼티와 생성자를 모두 가질 수 있도록 개선한 reader 빈을 살펴보자. componentName 프로퍼티는 setter 메서드를 이용했고, filename 인자는 생성자를 이용해 주입했다.

```
<bean name="reader"
      class="com.madhusudhan.jscore.fundamentals.injection.FileReader">
  <constructor-arg
    value="src/main/resources/basics/basics-trades-data.txt" />
  <property name="componentName" value="TradeFileReader" />
</bean>
```

이처럼 주입 방식을 혼용해서 사용할 수는 있지만 복잡한 문제가 발생하지 않도록 하려면 한 가지 방법만 사용하는 편이 좋다.

2.7 빈 콜백

스프링은 콜백 메서드의 형태로 두 가지 후크 메서드를 빈에 제공한다. 이 메서드는 빈이 프로퍼티를 초기화하거나 리소스를 비우게 해줄 수 있으며, 두 개의 후크 메서드는 init-method와 destroy-method라는 속성에서 사용된다.

2.7.1 init-method 속성

빈을 생성할 때 초기화를 위해 스프링이 특정한 메서드를 호출하게 할 수 있다. 이 메서드는 빈으로 하여금 필요한 작업을 실행하게 하고 데이터 구조 및 스레드 풀을 생성하게 하는 등의 초기화 작업을 가능케 한다.

환율Foreign eXchange, FX을 가져오는 클래스를 생성하라는 요구 사항을 받았다고 가정하자. 다음 소개하는 FxRateProvider 클래스는 환율을 조회했을 때 해당 값을 제공하는 클래스다(환율을 계산하는 가상 구현물).

```
public class FxRateProvider {
  private double rate = 0.0;
  private String baseCurrency = "USD";
  private Map<String, Double> currencies = null;

  /* 스프링의 init-method 속성으로 호출할 콜백 메서드 */
  public void initMe() {
    currencies = new HashMap<String, Double>();
    currencies.put("GBP",1.5);
    currencies.put("USD",1.0);
    currencies.put("JPY", 1000.0);
    currencies.put("EUR",1.4);
    currencies.put("INR",50.00);
  }
  public double getRate(String currency) {
    if(!currencies.containsKey(currency))
      return 0;
    return currencies.get(currency);
  }
}
```

여기서 중요한 것은 initMe 메서드인데, 이는 빈 생성 과정에서 스프링이 호출하는 평범한 메서드이기 때문이다. 빈과 관련해서는 다음 메타데이터를 확인하자.

```
<bean name="fxRateProvider"
      class="com.madhusudhan.jscore.fundamentals.callbacks.FxRateProvider"
      init-method="initMe">
  <property name="baseCurrency" value="USD" />
</bean>
```

2.7.2 destroy-method 속성

스프링은 init-method 속성과 비슷한 destroy-method라는 속성을 이용해 빈을 파괴하기 직전에 사용할 수 있는 파괴 메서드를 제공한다. 다음 FxRateProvider 클래스는 파괴 메서드를 사용한 예다.

```
public class FxRateProvider {
  public void destroyMe() {
    // 애플리케이션 청소 작업을 여기서 하면 됨
    currencies = null;
  }
  ...
}
```

메타데이터를 설정할 때는 다음처럼 destroyMe 메서드를 참조해야만 한다.

```
<bean name="fxRateProvider"
      class="com.madhusudhan.jscore.fundamentals.callbacks.FxRateProvider"
      init-method="initMe"
      destroy-method="destroyMe">
  <property name="baseCurrency" value="USD" />
</bean>
```

프로그램을 종료할 때 스프링은 빈을 파괴한다. 즉, 스프링이 파괴 프로세스를 진행할 때 설정 메타데이터에 destroyMe 메서드를 destroy-method 속성으로 선언했으므로 destroyMe 메서드를 호출하는 것이다. 이 메서드 때문에 빈은 필요한 작업을 할 기회가 생긴 것이다. 이때 리소스를 해제하거나, 객체를 null로 만들거나, 다른 청소 작업을 하는 것이 좋다.

2.7.3 공통 콜백

모든 빈에 init-method와 destroy-method 속성을 사용하도록 했다고 생각해보자. 이는 각각의 빈 모두에 init-method와 destroy-method 속성을 명시적으로 선언해줘야 한다는 뜻일까? 그렇지는 않다.

모든 빈에 같은 이름의 메서드를 정의했다면 스프링에서 기본 콜백을 선언하게 하면 된다. 즉, default-init-method 속성과 default-destroy-method 속성을 사용하는 것이다.

따라서 개별 메서드를 각각의 빈에 선언하는 대신 최상위의 〈beans /〉 요소에 이 콜백을 선언할 필요가 있다. 다음 XML 소스 코드를 살펴보면서 이를 어떻게 선언하는지 알아보자.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd"
       default-init-method="initMe"
       default-destroy-method="destroyMe">
  <bean>
    ...
  </bean>
</beans>
```

위 기본 메서드는 〈beans /〉 요소 하나와 연계된 것이 아니라 여러 개의 〈bean /〉 요소와 연계되었다는 점을 주의하자. 또한 위 예제에 따르면 initMe와 destroyMe 메서드는 모든 빈에서 자동으로 호출할 것이며, 이러한 메서드를 가지지 않은 빈이 있다면 스프링은 이 메서드를 무시하고 아무 작업도 하지 않을 것이다.

2.8 요약

2장에서는 스프링을 좀 더 자세하게 살펴보았다. 빈의 개념과 빈 팩토리를 설명했고 빈 스코프의 라이프 사이클도 살펴봤다. 그리고 자바 컬렉션이나 다른 타입의 객체를 주입할 때 사용하는 프로퍼티 에디터도 간단히 언급했다.

3장에서는 스프링이 실제로 동작하는 데 있어 중요한 개념인 컨테이너와 애플리케이션 컨텍스트를 알아보도록 하자.

3 | 컨테이너

스프링 컨테이너는 스프링의 핵심이다. 여기서 컨테이너란 기본적으로 애플리케이션을 처음 실행할 때 스프링이 메모리 공간에 생성하는 빈Bean의 영역pool을 말하며, 스프링에서 제공하는 API는 컨테이너 빈을 조회할 수 있는 메서드를 가진다. 3장에서는 컨테이너와 컨테이너 종류에 따른 차이뿐만 아니라 오토와이어링AutoWiring과 같은 개념도 자세히 살펴볼 것이다.

3.1 컨테이너

컨테이너는 스프링에서 매우 중요한 부분(직소 퍼즐의 움직이는 조각처럼)이므로, 스프링으로 작업을 시작하기 전에 먼저 컨테이너부터 이해해두어야 한다.

프로그램이 시작될 때 빈은 컨테이너 안에서 인스턴스화하고, 연관성과 관련성을 만들며, 모든 관계성을 충족하고, 의존성dependency을 주입inject한다. 그리고 모든 빈은 API를 사용해서 조회하는 일이 가능하다. 또한 지연 로딩lazily loaded 기법으로 필요한 시점에 컨테이너에 로드하는 빈도 있고 컨테이너를 초기화할 때 함께 로드하는 빈도 있다. 즉, 직접 혹은 의존성의 한 부분으로 애플리케이션이나 다른 빈이 해당 빈을 호출하지 않는 이상, 스프링은 빈을 인스턴스화하지 않는다는 뜻이다. 이는 싱글턴singleton 기법으로 로드할 때를 제외하면 스프링의 모든 컨테이너에 똑같이 적용된다.

스프링 컨테이너는 주로 빈 팩토리와 애플리케이션 컨텍스트라는 두 종류로 분류하며, 이는 BeanFactory와 ApplicationContext 인터페이스를 사용한다. 사실 이름만으로는 컨테이너인지 혹은 무슨 일을 하는지 유추하기가 어려우므로 잘 지어진 이름이라고 보기는 어렵다.

일반적으로 빈 팩토리는 기본적인 의존성 주입Dependency Injection, DI을 지원하는 간단한 컨테이너이며, 애플리케이션 컨텍스트는 이에 몇 가지 기능을 추가해 확장한 컨테이너라고 이해하자.

필자는 빈 팩토리보다는 애플리케이션 컨텍스트를 선호하는 편이며, 그 이유는 나중에 설명하겠다.

3.1.1 빈 팩토리 컨테이너

빈 팩토리 컨테이너는 스프링이 제공하는 두 가지 컨테이너 중 가장 단순한 유형이며, org.springframework.beans.factory.BeanFactory 인터페이스를 구현해야만 한다. 빈 팩토리 컨테이너의 핵심은 빈을 생성하고 인스턴스화할 때 모든 의존성을 함께 설정한다는 점이다. 즉, 빈을 조회했을 때 모든 연관성과 관계성이 이미 결정되어 완전히 제 기능을 할 수 있는 상태의 인스턴스를 얻을 수 있다는 뜻이다. 따라서 빈 팩토리 컨테이너는 사용자화한 메서드를 init-method와 destroy-method 속성으로 호출할 수 있다. 또한 스프링은 특별히 빈 팩토리 컨테이너에 몇 가지 클래스를 구현해 놓았는데, 그중 가장 많이 사용하는 클래스는 XmlBeanFactory로, 이름에서도 알 수 있듯이 XML 파일에서 메타데이터로 설정한 빈의 기본 정보를 읽는 클래스다.

XXXClient 클래스에서 XmlBeanFactory 클래스를 사용하는 방법은 이전에 애플리케이션 컨텍스트 컨테이너를 사용한 예제에서 살펴봤던 것과 비슷하므로 쉽게 이해할 수 있을 것이다. 다음 소스 코드는 factory 객체의 인스턴스화를 보여준다. 여기서 생성자는 XML 파일 이름을 인자로 받아 FileInputStream 클래스 객체로 전달한다.

```
// 빈 설정 파일을 이용해 factory 객체로 인스턴스화함
BeanFactory factory
  = XmlBeanFactory(new FileInputStream("trade-beans.xml"));
```

```
// TradeService 빈을 얻으려고 factory 객체를 이용함
TradeService service = (TradeService) factory.getBean("tradeService");
```

빈 팩토리 컨테이너는 보통 핸드폰 등과 같이 리소스가 한정된 작은 디바이스의 애플리케이션에서 주로 사용된다. 표준 자바나 JEE 애플리케이션을 사용한다면 빈 팩토리보다는 애플리케이션 컨텍스트 컨테이너를 구현하는 편이 이상적이다. 그러므로 특별한 이유가 아니라면 애플리케이션 컨텍스트를 사용하는 편이 좋다.

3.1.2 애플리케이션 컨텍스트 컨테이너

애플리케이션 컨텍스트 컨테이너는 빈 팩토리 컨테이너의 확장 버전으로, 따라서 빈 팩토리 컨테이너의 모든 기능은 이미 애플리케이션 컨텍스트 컨테이너에 내장되어 있다. 또한 추가로 애플리케이션 이벤트와 같은 고급 기능도 제공한다. 이런 확장성 때문에 애플리케이션 컨텍스트 컨테이너 하나만 구현해도 괜찮다.

빈 팩토리와 마찬가지로 애플리케이션 컨텍스트 컨테이너는 다음 세 가지 클래스를 구현했다.

- **FileSystemXmlApplicationContext 클래스:** 지정한 파일 시스템에 있는 XML 파일에 작성한 빈 정의definition를 읽는다. 이를 위해서는 반드시 이 파일의 전체 경로를 생성자에게 제공해야 한다. 즉, 애플리케이션이 소스 코드 자체에 설정 내용을 가진 때보다 특정 위치에 존재하는 XML 파일을 로드할 때 주로 사용된다.

- **ClassPathXmlApplicationContext 클래스:** XML 파일을 클래스 경로classpath에서 가져온다. 해당 XML 파일이 애플리케이션의 클래스 경로에 있다면 경로가 어디든지 이를 로드할 수 있으며, 빈 파일을 jar 파일 형태로 가질 수도 있다. FileSystemXmlApplicationContext 클래스와 유일하게 다른 점

은 파일의 전체 경로가 필요하지 않다는 것이다. 하지만 애플리케이션의 클래스 경로는 필요하다.

- **WebXmlApplicationContext 클래스:** 웹 애플리케이션 안의 모든 빈 정의를 기록한 XML 파일을 가져온다. 이 클래스는 주로 스프링 MVC 프로젝트나 자바 웹 애플리케이션에서 사용된다.

1장에서 이미 애플리케이션 컨텍스트 타입의 컨테이너 사용 방법을 살펴봤지만 다시 한번 복습해보자.

이전에 이야기했듯이 ClassPathXml ApplicationContext 클래스는 빈을 정의한 XML 파일을 로드해 같이 인스턴스화한다. 다음 소스 코드는 containers-beans.xml 파일을 로드해 컨테이너를 생성한다.

```
ApplicationContext ctx =
  new ClassPathXmlApplicationContext("containers-beans.xml");
Employee employee = ctx.getBean("employee",Employee.class);
```

다음 소스 코드는 employee 빈을 조회하는 메커니즘을 보여주는 예로, 여러 개의 XML 파일에서 컨테이너를 생성할 때는 String 배열을 이용해 생성자를 인스턴스화하면 된다.

```
public class ApplicationContextClient {
  private String[] configLocations
    = new String[]{"containers-beans.xml","fundamentals-beans.xml"};

  public void testClasspathXmlApplicationContext() {
    ctx = new ClassPathXmlApplicationContext(configLocations);

    Employee employee = ctx.getBean("employee",Employee.class);
```

```
    Person person = ctx.getBean("person", Person.class);
    System.out.println(employee);
    System.out.println(person);
  }
  ...
}
```

컨테이너를 생성하는 방법은 여러 가지가 있기 때문에, 이에 대해서는 API 문서를 확인해보는 편이 좋다. 예를 들어 FileSystemXmlApplicationContext 클래스는 ClassPathXmlApplicationContext 클래스와 거의 같지만 파일 경로를 통해 설정 파일을 가져온다. 이처럼 방법이 조금씩 다르므로 여러 방법을 이용할 수 있다.

다음 소스 코드는 FileSystemXmlApplicationContext 클래스로 컨테이너를 생성하는 메커니즘을 보여주는 예다.

```
ApplicationContext ctx
  = new FileSystemXmlApplicationContext
    ("/src/main/resources/containers-beans.xml");
Employee employee = ctx.getBean("employee",Employee.class);
```

다양한 종류의 컨테이너 중에서 어떤 것을 선택해야 좋은지는 해당 조직의 애플리케이션 배포나 환경 설정 방법에 따라 다르다. 어떤 조직에서는 모든 XML 파일을 한군데 모으는 경향이 있을 수 있는데, 이때는 애플리케이션을 새로 빌드하거나 설치하지 않았는데도 XML 파일을 변경하거나 수정할 수도 있다. 하지만 파일 위치는 변하지 않으므로 FileSystemXmlApplicationContext 클래스로 컨테이너를 생성해 사용하는 편이 가장 좋다.

어떤 프로젝트는 한군데서 설정 데이터를 읽어오는 것이 아니라 프로젝트 자체에 XML 파일이 함께 붙어있거나 jar 파일로 함께 묶여있을 수 있다.

이런 경우 ClassPathXmlApplicationContext 클래스로 컨테이너를 생성하고 사용하는 것이 가장 좋은 선택이라고 할 수 있다. 이미 눈치챘을 수도 있지만 이 책에서는 ClassPathXmlApplicationContext 클래스를 가장 많이 사용했다.

3.2 빈 인스턴스화

스프링에서 빈을 인스턴스화하는 방법은 여러 가지며 2장에서는 생성자를 사용해 빈을 인스턴스화하는 방법을 설명했다. 이는 빈의 생성자와 그에 대응하는 XML 파일의 메타데이터를 제공해야만 하고, 생성자가 인자를 가졌다면 XML 파일의 〈constructor-arg /〉 요소에 해당 인자를 입력하면 된다는 뜻이다.

이제는 그 외의 빈을 인스턴스화하는 두 가지 방법인 정적 메서드와 팩토리를 이용하는 방법을 알아보자.

3.2.1 정적 메서드 이용

인스턴스를 생성하려고 클래스에서 정적 메서드를 사용해야만 하는 때가 있는데, 지금 바로 떠오르는 게 자바 싱글턴 패턴이다. 스프링은 생성자를 통하는 것이 아니라 노출된 정적 메서드를 사용해 인스턴스를 생성할 수 있는 메커니즘을 제공한다. 클래스에서 객체 인스턴스를 생성할 때 정적 팩토리를 사용하는 경우라면 이러한 방식으로 인스턴스화하는 편이 좋다. 절차는 매우 간단하다. 해당 클래스의 인스턴스를 생성하는 정적 메서드를 가진 클래스를 만들면 된다.

다음 소스 코드는 private 생성자와 정적 메서드의 인스턴스 생성자를 가진 자바 싱글턴 패턴을 따르는 EmployeeFactory 클래스를 보여주는 예다.

```
public class EmployeeFactory {
  private static EmployeeFactory instance;
  private EmployeeFactory() { }
  public static EmployeeFactory getEmployeeFactory() {
```

```
    if (instance == null) {
      instance = new EmployeeFactory();
    }
    return instance;
  }
  ...
}
```

정적 메서드인 getEmployeeFactory는 이 클래스의 인스턴스를 반환한다. 이제 정적 메서드를 가진 클래스를 준비했으니, 다음 단계로 XML 파일에 빈을 정의해야 한다. 빈 정의는 factory-method 속성을 추가하는 일을 제외하고는 예전에 했던 방식과 비슷하다.

그럼 factory-method 속성을 사용해서 XML 파일에 employeeFactory 빈을 선언하는 방법을 알아보자.

```
<bean name="employeeFactory"
      class="com.madhusudhan.jscore.containers.factory.EmployeeFactory"
      factory-method="getEmployeeFactory">
</bean>
```

factory-method 속성은 클래스 각각의 정적 메서드를 호출하며, EmployeeFactory 클래스의 인스턴스는 factory-method 속성의 getEmployeeFactory 메서드를 호출해 인스턴스화한다.

물론 위에서 이 속성 선언 어느 부분에도 정적화static하라는 언급은 없었지만, 정적화하지 못해 발생하는 에러를 피하려면 반드시 getEmployeeFactory 메서드를 static으로 선언해주는 편이 좋다.

또한 반환 타입 메서드도 꼭 사용할 필요가 없으므로(반환 시 non-void 타입 값을 반환해야만 한다) 해당 클래스에 구현한 메서드를 직접 확인하지 않는 이상 알기 어렵다는 점을 기억해야 한다.

3.2.2 팩토리 메서드 이용

바로 앞에서 여러분은 정적 메서드를 이용해 객체를 생성했다. 그럼 정적 메서드를 사용하지 않고 인스턴스를 생성해야 할 때는 어떻게 해야 할까? 스프링은 일반non-static 팩토리 메서드를 이용해 빈을 인스턴스화하는 방법도 제공한다. 단순한 방법은 아니지만 이해하기 어려울 정도도 아니다.

다음 EmployeeCreator 클래스는 두 개의 팩토리 메서드인 createEmployee 메서드와 createExecutive 메서드를 생성하는 간단한 클래스다.

```
public class EmployeeCreator {
  public Employee createEmployee() {
    return new Employee();
  }
  public Employee createExecutive() {
    Employee emp = new Employee();
    emp.setTitle("EXEC");
    emp.setGrade("GRADE-A");
    return emp;
  }
}
```

팩토리 메서드를 이용할 때의 핵심은 XML 파일에 있다.

다음 소스 코드에서 employeeCreator 빈은 평범한 빈이다. 하지만 employee와 executive 빈은 factory-bean과 factory-method라는 두 개의 속성을 이용했다.

자세한 내용은 다음 소스 코드를 살펴보도록 하자.

```
<bean name="employee"
      factory-bean="employeeCreator"
      factory-method="createEmployee" />
<bean name="executive"
      factory-bean="employeeCreator"
      factory-method="createExecutive" />

<!-- 위 두 빈은 EmployeeCreator 클래스를 이용했다 -->
<bean name="employeeCreator"
      class="com.madhusudhan.jscore.containers.factory.EmployeeCreator" />
```

employee와 executive 빈의 factory-bean 속성은 모두 employeeCreator 빈을 참조한다. 이는 정적이 아닌 팩토리 메서드를 이용해 작업할 때 따라야 할 규약이다. 즉, factory-method 속성은 팩토리 빈인 employeeCreator의 실제 메서드를 가리키는데, 이때는 EmployeeCreator 클래스의 createEmployee와 createExecutive 메서드를 가리킨다.

여기에 클래스 속성을 전혀 포함하지 않았다는 점을 눈치챘는가? 생략한 부분은 factory-bean 속성이 참조하는 실제 빈과 factory-method 속성이 각각 참조하는 실제 메서드다. 참고로, 위 예제에서 factory-method 속성은 정적 메서드로 구현하지 않았다.

3.3 콜백의 초기화와 파괴

2장에서 빈의 초기화와 파괴를 담당하는 init-method 속성과 destroy-method 속성을 배웠다.

이 속성에서 지정한 메서드는 빈의 라이프 사이클에서 여러분에게 필요한 내부 작업을 할 수 있게 도와준다. 이미 알았겠지만, 이 속성은 클래스의 소스 코드가 아닌 설정 파일에서 사용한다.

하지만 이러한 역할을 설정 파일에서가 아니라 프로그램에서 구현해야 할 때도 있다. 그럴 때 스프링은 InitializingBean과 DisposableBean이라는 두 개의 인터페이스를 클래스에 구현함으로써 이러한 기능을 사용할 수 있게 한다. 인터페이스에는 각각 afterPropertiesSet 메서드와 destroy 메서드가 존재한다.

3.3.1 InitializingBean 인터페이스의 afterPropertiesSet 메서드

한 주에 오직 3일만 일하기를 원하는 LazyEmployee 클래스가 있고 생각해보자. lazyEmployee 빈 정의에서 지정한 weekDays 변수에 이러한 상태를 반영한다.

빈 정의는 다음과 같다.

```
<bean name="lazyEmployee"
      class="com.madhusudhan.jscore.containers.lifecycle.LazyEmployee">
  <property name="weekDays" value="THREE" />
</bean>
```

빈을 초기화했을 때 해당 변수 값은 예상한 대로 THREE(3일)로 지정된다. 하지만 이미 빈을 생성한 후에 누군가가 이 기능을 다시 FIVE(5일)로 정의하려면 어떻게 해야 할까(예를 들면 이 개발자의 상사라든지!)? 바로 이때 InitializingBean 인터페이스의 afterPropertiesSet 메서드를 사용할 수 있다.

여기서 해야 할 일은 LazyEmployee 클래스에 InitializingBean 인터페이스를 구현하는 것뿐이다. 다음 소스 코드를 살펴보자.

```
public class LazyEmployee implements InitializingBean {
  private String weekDays = null;

  public void afterPropertiesSet() throws Exception {
    System.out.println("AfterPropertiesSet called");
    weekDays = "FIVE";
  }
  ...
}
```

인터페이스는 오직 afterPropertiesSet 메서드 하나만 가지며, 이 메서드를 구현해야 한다. 또한 해당 메서드는 모든 프로퍼티를 설정하고 빈을 생성한 다음에 호출하는데, 여기서 프로퍼티 값들은 모두 XML 파일에서 가져온다는 점을 유의하자.

이 예제는 XML 파일에서 weekDays 값이 THREE로 지정되어 있다. 하지만 이 값을 가지고 이미 생성한 빈을 넘기기 전에 스프링은 해당 값을 FIVE로 변경하는 afterPropertiesSet 메서드를 호출한다. 즉, InitializingBean 인터페이스를 구현하면 빈의 설정을 초기화하거나 재설정할 기회를 제공하는 것이다.

3.3.2 DisposableBean 인터페이스의 destroy 메서드

DisposableBean 인터페이스는 destroy라는 콜백 메서드 하나만 가지는데, 이는 스프링 컨테이너를 종료하기 직전 호출하므로 필요할 때 리소스를 해제할 수 있다.

LazyEmployee 클래스를 계속 살펴보자. 이번엔 DisposableBean 인터페이스와 destroy 메서드를 구현해보겠다.

```
public class LazyEmployee implements DisposableBean {
  private String weekDays = null;

  public void destroy() throws Exception {
    System.out.println("Destroy called");
    // 애플리케이션 청소 작업을 여기서 하면 됨
  }
  ...
}
```

ClassPathXmlApplicationContext 클래스의 close 메서드를 호출해 컨텍스트를 종료할 때가 되면 destroy 메서드를 호출한다. 그러므로 여기서 빈이 사라지기 전에 필요한 작업을 할 기회를 얻게 된다.

3.3.3 선언형 혹은 프로그램형 콜백

이제 빈을 초기화하거나 파괴할 때 두 가지 방법을 사용할 수 있음을 알게 되었다. 다시 한번 설명하면 init-method 속성이나 destroy-method 속성을 XML 파일에 선언해 사용할 수도 있고, InitialzingBean 인터페이스의 afterPropertiesSet 메서드나 DispozableBean 인터페이스의 destroy 메서드를 클래스에 구현해 같은 결과를 얻을 수 있다.

그런데 한 가지 더 이 작업을 할 수 있는 방법이 있다. 바로 @PostConstruct와 @PreDestroy 어노테이션을 사용하는 것이다. 4장에서 어노테이션을 사용하는 방법을 배울 때 이 어노테이션을 살펴볼 기회가 있을 것이다.

그럼 이들 중 어떤 방법을 이용하는 것이 좋을까? 개인적으로는 XML 파일을 이용하는 방법을 좋아한다. 프로그램형 콜백은 스프링의 인터페이스를 사용했기 때문에 소스 코드는 제3자에게 의존성을 가지도록 결합돼 있다. 이는 나중에 프로그램을 변경하는 일 등이 발생할 경우 많은 제약 사항을 초래한다. 예를 들어 다른 의존

성 주입 프레임워크로 변경하기를 원할 때(예를 들어 구글의 가이스Guice [01]) 이러한 의존성을 없애려면 소스 코드를 리팩토링해야 한다. 이전에 이야기했듯이 벤더에 얽매이는 일은 가능한 한 피하는 편이 최선이라는 것을 기억하자.

스프링에서는 다음과 같은 세 가지 방식의 콜백을 제공한다. 하지만 모든 콜백을 프로젝트 하나에서 사용하려면 작은 차이지만 우선순위를 따라야 한다. 스프링에서는 세 가지 방식이 함께 존재할 때(어노테이션, 프로그램형, 설정 파일 콜백) 다음 순서대로 우선순위를 지정한다.

1. 어노테이션 기반: @PostConstruct과 @PreDestroy 어노테이션
2. 프로그램형 기반: afterPropertiesSet과 destroy 메서드
3. 설정 파일 기반: init-method와 destroy-method 속성

3.4 이벤트 핸들링

가끔은 사용자화 프로세스를 구현하기 위해 컨테이너 안에서 일어나는 이벤트에 반응할 필요가 있다.

예를 들어 컨테이너를 작동한 직후 FilePoller 컴포넌트가 수신 파일을 가져오려고 디렉터리를 폴링polling하기 원한다고 하자. 스프링에는 컨텍스트가 시작하거나 멈췄을 때 이를 알려주는 메커니즘이 있다. 이는 잠시 후에 알아보기로 하자.

또한 빈에서 메시지를 보내야 한다는 새로운 요구 사항이 있을 때도 있다. 예를 들어 FilePoller 컴포넌트가 빈에 파일을 받았는지 혹은 암호화 파일을 처리했는지 등의 내용을 전달해야 할 필요가 있을 수 있다. 스프링은 이러한 사용자화 이벤트를 처리하거나 발생시키는 방법도 제공한다.

---

01 http://code.google.com/p/google-guice

애플리케이션 컨텍스트 컨테이너는 빈을 로드할 때 지정한 유형의 이벤트 몇 개를 발생시킨다. 예를 들면 컨텍스트가 시작할 때는 ContextStartedEvent 클래스로 이벤트를 발생시키고 컨텍스트가 멈출 때는 ContextStoppedEvent 클래스로 이벤트를 발생시킨다. 이러한 이벤트를 바탕으로 둔 무언가를 구현해야 할 때 빈이 이러한 이벤트를 받게 할 수 있으며, 직접 구현한 이벤트를 발생시킬 수도 있다.

우선 이벤트에 반응할 때의 절차를 알아보자.

3.4.1 컨텍스트 이벤트 수신

애플리케이션 컨텍스트는 빈을 로드할 때 특정 이벤트를 발생시킨다. 앞에서 설명한 것처럼 컨텍스트를 시작할 때는 ContextStartedEvent 클래스로 이벤트를 발생시키고 컨텍스트를 종료할 때는 ContextStoppedEvent 클래스로 이벤트를 발생시킨다. 또한 빈이 이벤트를 받아 이를 기반으로 필요한 작업을 하게 할 수도 있다.

빈에서 컨텍스트 이벤트에 반응하려면 ApplicationListener 인터페이스를 구현해야만 한다. 이 인터페이스는 오직 onApplicationEvent라는 메서드만 가진다. ContextStartedEventListener는 컨텍스트의 시작과 관련한 이벤트를 받는 클래스며, 다음은 ContextStartedEventListener 클래스를 생성하는 예제다.

```
public class ContextStartedEventListener
  implements ApplicationListener<ContextStartedEvent> {
  public void onApplicationEvent(ContextStartedEvent event) {
    System.out.println(
      "Received ContextStartedEvent application event:"+event.getSource()
    );
  }
}
```

ContextStartedEvent 클래스로 이벤트를 발생시킬 때마다 빈은 이 이벤트를 받을 것이다. 그리고 현재 onApplicationEvent 메서드는 이벤트를 받았을 때 이를 알리는 것 이외에는 아무 작업도 하지 않고 있지만 여기에 다른 작업을 구현할 수도 있다. 예를 들어 새로운 컴포넌트를 시작한다든지, 리소스 로더를 생성한다든지, 이메일을 보낸다든지 등과 같은 작업을 구현할 수 있다.

이제 이벤트 리스너를 완성했으니 다음에는 리스너를 컨텍스트 이벤트와 연결해야 한다. 이를 위해 XML 파일에 빈을 선언한다.

```
<bean name="contextStartEventListener" class
  ="com.madhusudhan.jscore.containers.event.ContextStartedEventListener" />
```

컨텍스트가 시작하면 예상대로 리스너가 ContextStartedEvent 클래스로 발생시킨 이벤트를 받는다.

스프링은 다음과 같은 클래스로 이벤트를 발생시킨다.

- **ContextStartedEvent 클래스:** ApplicationContext 컨테이너가 로드하기 시작할 때 발생시키며, 빈은 ApplicationContext 컨테이너가 로드하기 시작할 때 시작 신호를 받는다. 데이터베이스 폴링이나 파일 시스템 탐색과 같은 작업은 이러한 클래스로 이벤트를 받았을 때 시작할 수 있다.
- **ContextStoppedEvent 클래스:** ContextStartedEvent 클래스로 발생시키는 이벤트와는 반대로 ApplicationContext 컨테이너의 실행을 중지했을 때 이벤트를 발생시킨다. 이때 스프링은 빈에게 중지 신호를 보내므로 이를 받아 필요한 작업을 하면 된다.
- **ContextRefreshedEvent 클래스:** 컨텍스트를 초기화하거나 재설정했을 때 이벤트를 발생시킨다.

- **ContextClosedEvent 클래스**: ApplicationContext 컨테이너의 실행을 종료했을 때 이벤트를 발생시킨다. 종료한 상태에서 컨텍스트는 재시작하거나 새로 고침할 수 없다.

- **RequestHandledEvent 클래스**: 웹에서 요청web request을 받았을 때 이벤트 리시버에게 알려주는 web-specific 이벤트를 발생시킨다.

끝으로 컨텍스트를 시작하고 새로 고침한 후 종료하는 클라이언트를 실행해보자.

```
public class ContextEventListenerClient {
  private ConfigurableApplicationContext ctx = null;
  public void test() {
    ctx
      = new ClassPathXmlApplicationContext("containers-events-beans.xml");
    ctx.start();
    ctx.refresh();
    ctx.stop();
  }
  public static void main(String args[]) {
    ContextEventListenerClient client = new ContextEventListenerClient();
    client.test();
  }
}
```

리스너는 각 상황에 맞는 적절한 이벤트를 받는다.

3.4.2 사용자화 이벤트 발생

사용자화 이벤트를 발생시키는 일은 쉽다. 100MB가 넘는 파일을 받을 때마다 이벤트를 발생시킨다고 생각해보자. 애플리케이션이 사용자화 이벤트를 발생시킬

수 있게 하려면 이전에 설명한 발행자, 리스너, 이벤트, 클라이언트 이 모두의 소스 코드 구조를 좀 더 발전시켜야 한다.

그럼 개선 작업을 시작해보자. 우선 ApplicationEvent 클래스를 확장해 이벤트를 생성해야만 한다. 다음 HugeFileEvent 클래스를 살펴보자.

```
public class HugeFileEvent extends ApplicationEvent {
  private String fileName = null;
  public HugeFileEvent(Object source, String fileName) {
    super(source);
    this.fileName = fileName;
  }
}
```

이전에 말했듯이 이벤트 클래스는 ApplicationEvent 클래스에 기반을 두고 확장해야 한다.

이제 발행자를 생성할 차례다. 여기에서는 HugeFileEventPublisher라는 클래스를 생성한다. 이 클래스는 스프링의 ApplicationEventPublisherAware 인터페이스를 구현하므로 setApplicationEventPublisher 메서드를 호출하면 ApplicationEventPublisher 클래스 객체 pub를 클래스로 주입할 수 있다. 이때 HugeFileEventPublisher 클래스는 이벤트를 발생시킬 때 사용하는 publish 메서드를 가진다.

```
public class HugeFileEventPublisher implements
                                    ApplicationEventPublisherAware {
  private ApplicationEventPublisher pub = null;

  public void setApplicationEventPublisher(ApplicationEventPublisher pub) {
    this.applicationEventPublisher = pub;
```

```
    }

    public void publish(String fileName) {
      System.out.println("Publishing a HugeFileEvent, file is: "+fileName);
      HugeFileEvent hugeFileEvent = new HugeFileEvent(this,fileName);
      pub.publishEvent(hugeFileEvent);
    }
  }
```

세 번째 단계는 리스너 생성이다. 리스너가 없다면 이벤트를 발생시켜도 아무런 일을 할 수 없다(마치 짝사랑처럼 말이다!). 앞에서 리스너 구현을 설명한 바 있는데, 여기에서 HugeFileEventListener 클래스를 다시 한번 확인해보자.

```
  public class HugeFileEventListener implements
                                    ApplicationListener<HugeFileEvent> {
    public void onApplicationEvent(HugeFileEvent event) {
      System.out.println("Received Event:"+event.getSource());
    }
  }
```

이제 기본 클래스가 다 갖춰졌다. 남은 일은 이들을 연결하고 (XXXClient 클래스를 만들어) 호출하는 것이다.이를 위해 발행자인 hugeFileEventPublisher 빈과 리스너인 hugeFileEventListener 빈을 다음 소스 코드처럼 선언해야 한다.

```
  <bean name="hugeFileEventPublisher" class
  ="com.madhusudhan.jscore.containers.event.publish.HugeFileEventPublisher" />

  <bean name="hugeFileEventListener" class
  ="com.madhusudhan.jscore.containers.event.publish.HugeFileEventListener" />
```

이 코드가 동작할 수 있게 테스트 프로그램인 HugeFileEventClient 클래스를 만들어보자.

```
public class HugeFileEventClient {
  private ApplicationContext ctx = null;
  private HugeFileEventPublisher hugeFileEventPublisher = null;

  public void test() {
    ctx = new ClassPathXmlApplicationContext(
      "containers-events-publish-beans.xml"
    );
    hugeFileEventPublisher =
      ctx.getBean("hugeFileEventPublisher", HugeFileEventPublisher.class);
    hugeFileEventPublisher.publish("huge-file.txt");
  }
  public static void main(String args[]) {
    HugeFileEventClient client = new HugeFileEventClient();
    client.test();
  }
}
```

컨텍스트를 생성하면 발행자 빈을 가져와 발행 메서드를 호출해 이벤트를 발생시키게 된다. 이벤트를 발생시키면 이는 컨텍스트로 보내지며, 리스너는 해당 이벤트를 받아(스프링이 이벤트를 모든 컨텍스트 리스너에 전달해준다) 적절한 처리 작업을 한다.

3.4.3 싱글 스레드 이벤트 모델

스프링의 이벤트 처리와 관련해서 가장 중요하게 생각해야 할 부분은 스프링의 이벤트 처리를 싱글 스레드single-threaded로 처리한다는 것이다. 이때 기본적으로 동기 방식을 사용한다. 즉, 이벤트를 발생시켰을 때 수신자가 메시지를 받기 전에는

모든 프로세스가 그 상태에 멈춰있으며 더는 진행하지 않는다는 말이다. 따라서 이벤트 하나에 여러 개의 리스너가 반응하길 원한다면 싱글 스레드 모델은 애플리케이션의 성능 저하를 초래할 수 있다. 이벤트 처리를 사용해야 하는 애플리케이션을 설계할 때는 이 점에 특히 주의하자.

스프링의 이벤트는 싱글 스레드일 뿐만 아니라 스프링에 얽매이게 하므로 개인적으로는 이에 의존하지 않는 편이다. 왜냐하면 스프링의 인터페이스를 구현해야만 하므로 나중에 다른 프레임워크를 구현하려고 하면 일이 더욱 어려워지게 되기 때문이다. 이 외에도 이벤트 버스나 이벤트 리스너 프레임워크, 스프링 통합 프레임워크(간단한 이벤트 메커니즘에는 스프링 통합 프레임워크나 JMS는 너무 과분하긴 하지만)와 같은 오픈 소스에 기반을 둔 선택 사항이 많다.

스프링 통합에 관심을 둔다면 이미 출간한 『Just Spring Integration』(『스프링 인티그레이션 핵심 노트』로 출간할 예정)을 읽으면 도움이 될 것이다.

3.5 오토와이어링

기존에는 빈을 생성할 때 <property />나 <constructor-arg /> 요소를 이용해 프로퍼티를 지정하는 방법을 사용해왔다. 하지만 스프링은 입력한 내용을 저장하기 위해 관계성과 의존성을 자동으로 묶는 복잡한 개념을 내포하고 있다. 이 말은 즉, 프로퍼티와 프로퍼티 값을 명확하게 언급하지 않더라도 autowire 속성을 지정함으로써 스프링 이 해당 값들을 적절한 프로퍼티에 연결할 수 있게 해준다는 뜻이다. 이 메커니즘을 오토와이어링이라고 부른다.

오토와이어링은 기본적으로는 사용할 수 없게 되어 있다(autowire 속성을 no로 지정하는 것과 같다). 또한 오토와이어링은 오직 참조에만 사용해야 한다. 자바의 Primitive나 String 타입은 오토와이어링을 사용하도록 지정할 수 없기 때문이다.

오토와이어링에는 기본으로 세 가지 방식이 있다. 곧 설명할 byName, byType, byConstructor가 있으며, 오토와이어링과 명시적 와이어링의 혼용까지 생각한다면 총 네 가지 방식이 있다.

3.5.1 byName 오토와이어링

byName 오토와이어링을 사용하게 되면 스프링은 프로퍼티 필드와 이름을 맞춰봄으로써 적절한 의존성을 주입하려고 한다. 이는 예제를 살펴보면 쉽게 이해할 수 있을 것이다.

두 개의 프로퍼티 tradePersistor와 tradeTransformer를 가진 tradeReceiver 빈이 있다고 가정하자. byName을 사용하는 오토와이어링 메커니즘을 선택했다면 스프링은 의존성이 있는 빈을 컨텍스트에서 찾아(그들이 tradeReceiver 빈 안에 있는 프로퍼티와 같은 이름일 때만) 이를 tradeReceiver 빈으로 주입할 것이다.

먼저 다음 예제에 구현한 TradeReceiver 클래스의 정의를 살펴보자.

```
public class TradeReceiver {
  private TradePersistor tradePersistor = null;
  private TradeTransformer tradeTransformer = null;
  ...
}
```

보통 앞에서 소개한 이 세 개의 빈을 XML 파일에 정의한 후에 tradePersistor와 tradeTransformer 변수의 참조 값을 TradeReceiver 클래스로 전달하는 것이 일반적이다. 하지만 오토와이어링을 이용하면 이렇게까지 할 필요가 없다.

byname 오토와이어링을 사용하려면 빈 정의의 autowire 속성을 byName으로 지정해야 한다.

```
<bean name="tradeReceiver"
      class="com.madhusudhan.jscore.containers.autowire.TradeReceiver"
      autowire="byName" />

<bean name="tradePersistor"
      class="com.madhusudhan.jscore.containers.autowire.TradePersistor" />

<bean name="tradeTransformer"
      class="com.madhusudhan.jscore.containers.autowire.TradeTransformer" />
```

또한 tradePersistor와 tradeTransformer 빈을 XML 파일에 정의해야만 하는데, 이때 tradeReceiver 빈에 tradePersistor와 tradeTransformer 같은 프로퍼티를 전혀 정의하지 않았다는 것을 눈치챘는가? 해당 프로퍼티는 다음처럼 명시적으로 선언해주기만 했다.

```
<bean name="tradeReceiver"
      class="com.madhusudhan.jscore.containers.autowire.TradeReceiver">
  <property name="tradePersistor" ref="tradePersistor" />
  <property name="tradeTransformer" ref="tradeTransformer" />
</bean>
```

그렇다면 어떻게 동작하는 것일까? autowire="byName"은 보이지 않는 곳에서 마법을 부린다. 해당 값은 컨테이너로 하여금 tradeReceiver라는 변수 이름과 같은 이름을 가진 두 개의 프로퍼티를 찾게 한다(이 예제에서는 〈property /〉 요소의 name 속성 값으로 지정한 tradePersistor와 tradeTransformer다). 같은 이름을 가진 프로퍼티를 찾으면 이 빈을 바로 주입한다. 하지만 아무 프로퍼티도 찾

지 못했다면 빈 생성에 문제가 있다는 예외 상황이라는 메시지를 보낸다. 이러한 오토와이어링 원칙은 오직 참조할 때만 사용하자. 어차피 여러분은 오토와이어링과 명시적 와이어링을 혼용해야 한다(이러한 와이어링 방식의 혼용은 뒤에서 살펴볼 것이다).

3.5.2 byType 오토와이어링

byName 오토와이어링을 사용할 때와 마찬가지로 byType 방식의 오토와이어링을 사용하려면 autowire 속성을 지정해야 한다. 이때는 같은 이름으로 빈을 찾는 것이 아니라 같은 타입으로 찾는다. 66쪽에서 살펴봤던 tradeReceiver 빈 예제의 autowire 속성을 byType으로 바꾸어 지정하면 컨테이너는 TradePersistor나 TradeTransformer와 같은 타입의 빈을 찾는다.

```
<bean name="tradeReceiver"
      class="com.madhusudhan.jscore.containers.autowire.TradeReceiver"
      autowire="byType" />
```

컨테이너가 적절한 타입의 빈을 찾으면 해당 빈에 주입하게 된다. 위 예제에서라면 스프링은 com.madhusudhan.jscore.containers.autowire.TradePersistor와 com.madhusudhan.jscore.containers.autowire.TradeTransformer 같은 타입을 XML 파일에서 찾는다.

즉, 컨테이너가 적절한 타입을 찾으면 tradeReceiver 빈에 이를 주입할 것이다. 그러나 같은 타입을 가진 하나 이상의 빈을 찾을 경우에는 치명적인 예외 상황이 발생하게 된다.

3.5.3 constructor 오토와이어링

다음 오토와이어링이 무엇인지는 예상하기 쉽다. 바로 constructor 오토와이어링이다. 이는 byType 오토와이어링과 비슷하지만 빈의 생성자 인자에만 적용할 수 있다. 이름에서 알 수 있듯이 생성자 인자의 타입을 컨텍스트에서 찾을 것이다. 즉, 빈에서 다른 빈 타입의 인자를 취하는 생성자가 있을 때 컨테이너는 해당 참조 값을 찾아 이를 주입해야 한다는 뜻이다.

예를 들어 datasource 객체를 인자로 취하는 생성자를 가진 TradePersistor 클래스를 살펴보자.

```
public class TradePersistor {
  public TradePersistor (DataSource datasource) { .. }
}
```

위 예제에서 constructor 오토와이어링을 사용하면 컨테이너는 DataSource 타입의 객체를 찾아 이를 tradePersistor 빈으로 주입할 것이다. constructor 오토와이어링은 다음처럼 지정하면 사용할 수 있다.

```
<bean name="tradePersistor"
      class="com.madhusudhan.jscore.containers.autowire.TradeReceiver"
      autowire="constructor" />
```

3.5.4 명시적 와이어링과 오토와이어링의 혼용

사실 오토와이어링과 명시적 와이어링을 혼용하면 가장 이상적인 결과를 얻을 수 있다. 오토와이어링을 사용하면서 생기는 애매모호함은 명시적 와이어링으로 해결할 수 있기 때문이다.

예를 들어 다음 소스 코드를 살펴보면 tradeReceiver 빈은 두 개의 빈을 혼용해서 주입받는다. 이때 tradePersitor 빈은 예전과 마찬가지로 명시적으로 주입받으며 tradeTransformer 빈은 byName 오토와이어링을 사용해 자동으로 연결한다.

```
<bean name="tradeReceiver"
      class="com.madhusudhan.jscore.containers.autowire.TradeReceiver"
      autowire="byName">
  <!-- tradePersistor 빈은 이전에 했던 것처럼 평범하게 지정함 -->
  <property name="tradePersistor" ref="tradePersistor" />
</bean>
<bean name="tradePersistor"
      class="com.madhusudhan.jscore.containers.autowire.TradePersistor" />
<bean name="tradeTransformer"
      class="com.madhusudhan.jscore.containers.autowire.TradeTransformer" />
```

원한다면 특정 빈에 오토와이어링을 사용하지 않게 할 수도 있다. 다음 예제처럼 빈 정의에 autowire-candidate 속성 값을 false로 지정하면 된다. 여기서 tradePersistor 빈은 오토와이어링 모드에 포함하지 않을 것이다.

```
<bean name="tradePersistor"
      class="com.madhusudhan.jscore.containers.autowire.TradeReceiver"
      autowire-candidate="false" />
```

오토와이어링이 추가적인 메타데이터 선언을 줄여주기는 했지만 개인적으로는 좋아하지 않는 편이다. 이보다는 좀 더 명백하고 읽기 좋은 명시적 와이어링을 선호한다.

4 | 심화 개념

스프링은 여러 추가 기능을 가진다. 때때로 Map과 List 같은 자바 컬렉션을 애플리케이션에 주입할 필요도 있고, 빈을 위한 XML 파일을 사용자화한 내용으로 변경할 필요도 있다. 하지만 스프링에서는 이처럼 애플리케이션 외부의 컬렉션을 프로퍼티에 설정하는 일이나 빈을 원하는 대로 변경하는 일을 쉽게 할 수 있다. 이전 장까지는 스프링의 기본을 배웠다면 이제부터는 심화 개념을 알아보도록 하자.

4.1 빈 스코프

애플리케이션을 실행하면서 컨테이너가 XML 파일을 로드할 때 얼마나 많은 빈을 만드는지 생각해본 적 있는가? 혹은 매번 같은 빈 컨테이너를 조회할 때마다 같은 인스턴스를 얻게 될까? 컨테이너를 몇 번 호출하든 상관없이 단 하나의 빈만 만들어야 한다면 (서비스 빈이나 팩토리 빈 등을) 어떻게 해야 할까? 아니면 매 호출 시마다 새로운 빈을 인스턴스화하길 원하는가?

이러한 의문을 해결하기 위해 스프링은 무엇을 할까? 빈 생성을 지시하는 간단한 속성을 이용해 이러한 의문을 해결한다. 바로 singleton과 prototype 값을 가지는 scope 속성이다.

4.1.1 싱글턴 스코프

빈(이때는 trainFactory) 하나만을 원하는 경우라면 다음 소스 코드처럼 scope 속성 값을 singleton으로 설정하면 된다.

```
<bean name="trainFactory" scope="singleton"
    class="com.madhusudhan.jscore.fundamentals.scope.TrainFactory" />
```

하지만 기본 scope 속성 값은 항상 싱글턴이므로 다음처럼 설정하지 않아도 크게 상관은 없다.

```
<bean name="trainFactory"
      class="com.madhusudhan.jscore.fundamentals.scope.TrainFactory" />
```

이제 trainFactory 빈을 다른 빈으로 주입할 때나 getBean 메서드를 통해 조회할 때마다 같은 인스턴스를 반환할 것이다.

이때 자바 싱글턴 패턴을 사용할 때와 스프링 싱글턴을 사용할 때 얻어지는 인스턴스에는 미묘한 차이가 있음을 주의하기 바란다. 스프링 싱글턴은 컨텍스트나 컨테이너 각각의 싱글턴을 의미하지만, 자바 싱글턴은 프로세스나 클래스 로더 각각의 싱글턴을 의미한다.

4.1.2 프로토타입 스코프

scope 속성 값으로 prototype을 지정하면 빈을 호출할 때마다 새로운 인스턴스를 생성한다. 예를 들면 인스턴스화할 때마다 생성하는 Train 클래스 객체를 생성한다. 매번 새로운 Train 클래스 객체를 생성하는 일이 애플리케이션의 요구 사항이므로 XML 파일의 scope 속성 값은 prototype으로 지정해야만 한다. 그러면 호출할 때마다 새로운 Train 클래스 객체를 생성할 것이다.

```
<bean name="train" scope="prototype"
      class="com.madhusudhan.jscore.fundamentals.scope.Train">
  <property name="trainName" value="London to Paris Special" />
</bean>
```

다음 소스 코드를 살펴보면 알겠지만 클래스에서는 스코프에 대해 정의한 것이 하나도 없다. 즉, prototype 값이나 singleton 값을 사용하더라도 소스 코드에서 변

화하는 부분은 전혀 없다는 말이다. 스코프는 말 그대로 쓰인 대로 동작하므로 필요에 따라 이를 변경하면 된다.

마지막으로 train 변수와 trainFactory 변수로 해시 코드 값을 출력하는 테스트 클래스인 TrainClient를 살펴보자.

```
public class TrainClient {
  private static ApplicationContext context = null;
  private Train train = null;
  private TrainFactory trainFactory = null;

  public void init() {
    context = new ClassPathXmlApplicationContext("fundamentals-beans.xml");
  }

  private void checkTrainInstance() {
    for (int i = 0; i < 10; i++) {
      train = context.getBean("train", Train.class);
      System.out.println("Train instance: " + train.getInstance());
    }
  }

  private void checkTrainFactoryInstance() {
    for (int i = 0; i < 10; i++) {
      trainFactory = context.getBean("trainFactory", TrainFactory.class);
      System.out.println(
        "TrainFactory instance: " + trainFactory.getInstance()
      );
    }
  }

  public static void main(String[] args) {
```

```
        TrainClient client = new TrainClient();
        client.init();
        client.checkTrainInstance();
        client.checkTrainFactoryInstance();
    }
}
```

여기서 해시 코드 값을 확인하려고 소스 코드를 10번 실행하게 했다. singleton 값을 지정한 빈이라면 매번 같은 값이 나올 것이고, prototype 값을 지정한 빈이라면 다른 값이 나와야 한다. 따라서 이 TrainClient 클래스를 실행하면 다음과 같은 결과를 출력한다.

```
Train instance: 1443639316
Train instance: 975740206
Train instance: 1080513750
...
TrainFactory instance: 2074631480
TrainFactory instance: 2074631480
TrainFactory instance: 2074631480
...
```

앞에서 설명했듯이 Train 인스턴스는 매번 다른 값을 출력하고 TrainFactory 인스턴스는 같은 값을 출력한다.

마지막으로, 스프링에 기반을 두고 만들어진 웹 애플리케이션에서는 지금까지 설명한 singleton과 prototype 이외에도 scope 속성에 session이나 request, global-session과 같은 값을 지정할 수 있음은 기억해두자.

4.2 프로퍼티 파일

앞에서 FileReader 클래스나 FtpReader 클래스의 빈을 정의할 때 XML 파일에 직접 프로퍼티를 선언했었다. 하지만 파일 이름, FTP의 사용자 정보, 비밀 정보가 밀접하게 연결되어 있는 것은 결코 좋은 방법이 아니다.

왜냐하면 프로퍼티를 이전 환경에서 다른 환경으로 변경해야 할 때 이 XML 파일을 수정하는 일은 사실 좋은 방법이 아니기 때문이다. 그렇기에 스프링은 프로퍼티를 주입할 수 있는 다른 방법을 제공한다.

여기서 소개하는 방법은 name과 value 속성을 함께 사용하는 형태로 이루어진 프로퍼티 파일을 정의하고, 스프링의 PropertyPlaceholderConfigurer 클래스가 이를 읽게 하는 것이다. 이 클래스는 지정한 위치의 프로퍼티 파일을 로드해 프로퍼티를 확인한다. 이제 어떻게 확인하는지 살펴보자.

먼저 location과 type이라는 두 개의변수를 가진(두 개의 프로퍼티에서 사용할) JobSearchAgent 클래스를 생성하자. location 변수는 지원자가 선호하는 직장 위치를 의미하며, type 변수는 검색해야 할 직업의 종류(정규직, 임시직 등)를 의미한다.

```
public class JobSearchAgent {
  private String location = null;
  private String type = null;
  // 해당 프로퍼티의 setter와 getter 메서드 선언
  ...
}
```

이제 지원자가 속한 JobSearchAgent 클래스가 있다는 점을 고려해 두 개 프로퍼티를 외부 프로퍼티에서 로드해야 한다. 다음 단계는 리소스에 fundamentals-

beans.properties라는 프로퍼티 파일을 생성하고 다음 프로퍼티와 값을 추가하는 일이다.

```
job.location = /users/mkonda/dev/js;
job.type = permanent;
```

프로퍼티 파일의 이름을 지을 때 특정 규약을 따랐다는 점을 알아볼 수 있는가? 보통 프로퍼티 파일의 이름을 지을 때는 빈을 정의한 XML 파일에서 확장자만 제외한 채 그대로 쓰는 것을 선호한다(이전 예제에서 확장자는 .properties다).

마지막으로 확인해야 할 사항은 JobSearchAgent 클래스를 설정한 메타데이터다. fundamentals-beans.xml 파일에 PropertyPlaceholderConfigurer라는 스프링 클래스를 추가하도록 수정하자. 이 클래스는 여러분이 작성한 프로퍼티 파일의 위치를 가리키는 프로퍼티를 가진다.

```
<bean class
 ="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location"
            value="classpath:fundamentals-beans.properties" />
</bean>
```

PropertyPlaceholderConfigurer는 프로퍼티를 확인할 때 필요한 핵심 클래스다. 즉, 클래스 패스의 프로퍼티 파일을 찾아서 애플리케이션에 이를 로드한다. 마지막으로 jobSearchAgent 빈의 프로퍼티에 이를 매개변수화한다.

```
<bean name="jobSearchAgent"
      class="com.madhusudhan.jscore.fundamentals.properties.JobSearchAgent">
  <property name="type" value="${job.type}" />
  <property name="location" value="${job.location}" />
</bean>
```

${job.type}과 ${job.location} 프로퍼티의 파일 부분은 fundamentals-beans.properties 파일에 정의한 〈property /〉 요소의 name과 value 속성 부분에 해당하는 프로퍼티 값을 결정짓는다. 이들은 애플리케이션을 실행하면 적절하게 원하는 값으로 대체하므로 환경에 기반을 둔 프로퍼티를 생성할 수 있게 도와준다.

프로퍼티 파일을 지정할 때의 포맷은 유명한 Ant 스타일 패턴(${이름})을 따른다. 하지만 스프링은 접두사와 접미사를 변경할 수 있는 방법도 제공한다. 예를 들어 프로퍼티를 원래 스타일인 ${}이 아니라 #[이름]으로 사용하려면 name 속성에 placeholderPrefix 혹은 placeholderSuffix라는 값을 지정하고 value 속성 값에 원하는 방식을 지정하면 된다. 다음 소스 코드는 이를 보여주는 예다.

```
<bean class
 ="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location"
            value="classpath:fundamentals-beans.properties" />
  <property name="placeholderPrefix" value="#[" />
  <property name="placeholderSuffix" value="]" />
</bean>
```

물론 빈 프로퍼티에도 이러한 변경 사항을 적용해야 한다. 다음 소스 코드에서 볼드로 표시한 부분을 살펴보자.

```
<bean name="jobSearchAgent"
      class="com.madhusudhan.jscore.fundamentals.properties.JobSearchAgent">
  <property name="type" value="#[job.type]" />
  <property name="location" value="#[job.location]" />
</bean>
```

프로퍼티를 여러 경로에서 읽어와야 한다면 다음처럼 〈list /〉 요소를 이용해 프로퍼티 파일 리스트를 지정한다. 다음 locations 프로퍼티는 이를 보여주는 예다.

```
<bean class
 ="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
    <list>
      <value>classpath:fundamentals-beans.properties</value>
      <value>classpath:fundamentals-beans_1.properties</value>
    </list>
  </property>
</bean>
```

4.3 프로퍼티 에디터

XML 파일에 프로퍼티 속성 값을 지정할 때 선언 부분을 살펴보면 프로퍼티는 그저 String 타입이나 자바의 Primitive 타입처럼 보였다. 하지만 외부에서 초기화한 컬렉션이나 객체 참조 값 같은 프로퍼티도 존재한다.

스프링은 이러한 요구 사항에 대응하는 자바 빈 스타일의 프로퍼티 에디터 메커니즘을 따른다. 이를 이용하면 XML 파일에 선언하는 것만으로 List, Set, Map과 같은 자바 컬렉션을 주입하는 일이 가능하다.

또한 java.util.Properties 클래스도 제공하는데, 실제로 동작하는 방법은 곧 확인해볼 것이다. 프로퍼티가 다른 빈을 참조하는 참조 값이라면(의존성), 실제 빈 타입은 의존성을 주입할 때 결정한다.

4.3.1 자바 컬렉션 주입하기

스프링에서 List, Set, Map처럼 초기화해야 하는 컬렉션을 주입하는 일은 매우 쉽

다. XML 파일에서 지정한 문법을 따르기만 하면 주입하기 전 컬렉션을 초기화할 수 있기 때문이다.

List, Set, Map 사용하기

List는 자바 프로그램에서 자주 쓰는 컬렉션 중 하나다. 예를 들면 통화 목록이나 국가 목록, 평일 목록 등과 같이 이미 값을 할당한 리스트를 생성하는 경우다.

다음 CapitalCitiesManager 클래스에는 국가 목록을 string 타입 값으로 가지는 countriesList 변수를 선언했다. 이 변수는 값이 정해져 있으므로 XML 파일에 정의해 프로그램 외부에서 초기화할 수 있는 정적 List다.

```
public class CapitalCitiesManager {
  private List<String> countriesList = null;

  // getter와 setter 메서드 선언
  public List<String> getCountriesList() {
    return countriesList;
  }
  public void setCountriesList(List<String> countriesList) {
    this.countriesList = countriesList;
  }
}
```

List를 사용하려면 countrylist 프로퍼티(프로퍼티 이름은 변수 이름과 같아야 한다)를 〈list /〉 요소로 선언해야 한다. 〈list /〉 요소는 값을 가진 개별 〈value /〉 요소를 가진다.

다음은 CapitalCitiesManager 클래스에서 java.util.List 인터페이스를 어떻게 사용하는지 보여주는 메타데이터다.

```
<bean name="capitalCitiesManager" class =
"com.madhusudhan.jscore.fundamentals.propertyeditor.CapitalCitiesManager">
  <property name="countriesList">
    <list>
      <value>United Kingdom</value>
      <value>United States Of America</value>
      <value>India</value>
      <value>Japan</value>
    </list>
  </property>
</bean>
```

이와 비슷하게 Set를 이용해서 국가 목록을 생성하고 싶다면(단, java.util.Set 인터페이스는 같은 항목을 인정하지 않는다는 점에 유의하자) 위 예제의 〈list /〉 요소를 다음처럼 〈set /〉 요소로 변경하자.

```
<bean name="capitalCitiesManager" class =
"com.madhusudhan.jscore.fundamentals.propertyeditor.CapitalCitiesManager">
  <property name="countriesSet">
    <set>
      <value>United Kingdom</value>
      <value>United Kingdom</value>
      <value>United States Of America</value>
      <value>United States Of America</value>
      <value>India</value>
      <value>Japan</value>
    </set>
  </property>
</bean>
```

위 〈set /〉 요소에 중복 값을 일부러 만들어보았다. 중복 값을 제외한 네 개의 나라

만 Set에 할당한 것을 확인할 수 있다.

이번에는 Map을 살펴보자. 예상했겠지만 다음 소스 코드처럼 java.util.Map 인터페이스의 구현체를 사용해야 한다.

```
public class CapitalCitiesManager {
  private Map<String,String> countriesCapitalsMap = null;

  // getter와 setter 메서드 선언
  public Map<String, String> getCountriesCapitalsMap() {
    return countriesCapitalsMap;
  }
  public void setCountriesCapitalsMap(
   Map<String, String> countriesCapitalsMap) {
    this.countriesCapitalsMap = countriesCapitalsMap;
  }
}
```

하지만 Map을 설정할 때는 List나 Set과는 미묘하게 다른 부분이 있다. <map /> 요소에서는 name과 value 속성 대신 key와 value 속성으로 정의하는 엔트리 노드를 만든다는 점이다.

```
<bean name="capitalCitiesManager" class=
 "com.madhusudhan.jscore.fundamentals.propertyeditor.CapitalCitiesManager">
  <property name="countriesCapitalsMap">
    <map>
      <entry key="England" value="London" />
      <entry key="USA" value="Washington DC" />
    </map>
  </property>
</bean>
```

value 속성은 객체의 참조 값을 가질 수도 있다. 예를 들어 다음 예제에서 UK 엔트리는 UnitedKingdomUnion 클래스의 참조 값을 가졌다.

```
<bean name="capitalCitiesManager" class=
 "com.madhusudhan.jscore.fundamentals.propertyeditor.CapitalCitiesManager">
  <property name="countriesCapitalsRefMap">
    <map>
      <entry key="UK" value-ref="ukUnion" />
    </map>
  </property>
</bean>

<bean name="ukUnion" class
="com.madhusudhan.jscore.fundamentals.propertyeditor.UnitedKingdomUnion" />
```

여기서 ukUnion 빈은 이전에 사용했던 value 속성이 아니라 value-ref 속성을 사용해서 참조했다는 점을 주의하자.

4.3.2 자바 프로퍼티 주입하기

java.util.Properties 클래스 객체에도 name과 value 속성을 함께 사용할 수 있다. 먼저 CapitalCitiesManager 클래스에 countryProperties 변수를 선언한 소스 코드를 살펴보자.

```
public class CapitalCitiesManager {
  private Properties countryProperties = null;
  public Properties getCountryProperties() {
    return countryProperties;
  }

  public void setCountryProperties(Properties countryProperties) {
```

```
    this.countryProperties = countryProperties;
  }
}
```

다음에는 XML 파일의 빈에 연결할 차례다. 따라서 countryProperties 프로퍼티를 빈에 선언해야 한다.

```
<bean name="capitalCitiesManager" class=
 "com.madhusudhan.jscore.fundamentals.propertyeditor.CapitalCitiesManager">
  <property name="countryProperties">
    <props>
      <prop key="UK">London</prop>
      <prop key="USA">Washington DC</prop>
    </props>
  </property>
</bean>
```

countryProperties 프로퍼티는 위 소스 코드처럼 〈props /〉 요소로 나타낼 수 있으며, 각 요소는 〈prop /〉 요소의 key 속성과 값을 가진다.

4.4 빈 후처리기

스프링에서는 빈 후처리기bean post processor를 구현해 빈의 라이프 사이클에 관여할 수 있다. 후처리기에서는 설정 값을 변경하거나, 빈의 프로퍼티로 사용자화 값을 설정하거나, 초기화 프로세스를 좀 더 정교하게 재구현할 수도 있다.

스프링은 후처리기를 잘 활용하기 위해 BeanPostProcessor 인터페이스를 제공하며, postProcessBeforeInitialization와 postProcessAfterInitialization이라는 두 가지 콜백 메서드를 가진다. 또한 BeanPostProcessor 인터페이스는 컨테이너마다 설정한다.

즉, 해당 컨테이너의 모든 빈은 후처리기로 처리한다는 뜻이다. 원한다면 여러 가지 작업을 하기 위해 후처리기를 여러 개 가질 수도 있다.

이름에서도 알 수 있듯이 postProcessBeforeInitialization 메서드는 init-method 속성으로 afterPropertiesSet 메서드를 빈에서 호출하기 직전에 자동으로 호출한다. 빈을 막 실행하려는 순간이지만 그 사이에 마지막으로 무엇인가를 할 기회가 생긴 것이다.

postProcessAfterInitialization 메서드는 빈 초기화가 끝난 후에 호출한다. 간단한 예제를 살펴보자.

컨테이너에 선언한 빈에는 기본적으로 프로퍼티가 없다. 따라서 빈의 프로퍼티 역할을 해야 할 지정 변수(여기에서는 componentName)가 반드시 있어야 한다. 만약 componentName 변수가 없다면(비어있다면) 빈은 이를 반드시 기본 설정 값(예를 들어 LONDON-DEFAULT-COMP)으로라도 설정해야 한다. 이러한 조건을 충족할 수 있는 가장 쉬운 방법은 BeanPostProcessor 인터페이스를 구현하는 클래스를 생성하고 componentName 변수 값을 수정하는 것이다.

우선 DefaultComponentNamePostProcessor 클래스를 만들어보자.

```
public class DefaultComponentNamePostProcessor implements
                                                BeanPostProcessor {
  public Object postProcessBeforeInitialization(Object o, String string)
  throws BeansException {
    System.out.println("PostProcess BEFORE init: " +o);
    if (o instanceof PostProcessComponent) {
      ((PostProcessComponent)o).setComponentName("LONDON-DEFAULT-COMP");
    }
    return o;
  }
```

```
    public Object postProcessAfterInitialization(Object o, String string)
    throws BeansException {
      System.out.println("PostProcess AFTER init: " +o);
      if (o instanceof PostProcessComponent) {
        System.out.println(
          "Default Property: "+((PostProcessComponent)o).getComponentName()
        );
      }
      return o;
    }
  }
```

보다시피 DefaultComponentNamePostProcessor 클래스는 부모 클래스의 두 메서드(볼드체 부분)를 구현했다. 컨테이너에 생성한 빈은 바로 이 후처리기로 처리할 것이다. 그러므로 여기서 해야 할 일은 postProcessBeforeInitialization 메서드를 호출하는 동안 빈의 프로퍼티 역할을 해야 할 componentName 변수 값을 지정하고 postProcessAfterInitialization 메서드를 실행하는 동안 이 변수 값을 출력하는 것이다(그리고 수정하지 않은 인스턴스를 반환한다).

이를 완성하기 위해 간단한 POJO인 PostProcessComponent 클래스를 생성하고 아직 아무 값도 지정하지 않은 componentName 변수를 만들어보자.

다음은 PostProcessComponent 클래스의 소스 코드다.

```
  public class PostProcessComponent {
    private String componentName = null;
    public String getComponentName() {
      return componentName;
    }
    public void setComponentName(String componentName) {
```

```
        this.componentName = componentName;
    }
}
```

마지막으로 해야 할 일은 빈 후처리기와 빈을 XML 파일에 정의하는 것이다.

```
<bean name="compNamePostProcessor" class=
  "com.containers.postprocessor.DefaultComponentNamePostProcessor" />
<bean name="postProcessComponent" class=
  "com.madhusudhan.jscore.containers.postprocessorPostProcessComponent" />
```

XML 파일 어디에서든 빈을 후처리기로 선언할 필요 없이 보통의 빈으로만 선언해 주면 된다. 그러면 컨테이너는 추후 작업을 위해 모든 후처리기를 자동으로 탐색하고 불러들인다. 따라서 위 예제를 실행하면 다음 결과를 확인할 수 있다.

```
PostProcess BEFORE init:
  PostProcessComponent{componentName=null}
PostProcess AFTER init:
  PostProcessComponent{componentName=LONDON-DEFAULT-COMP}
```

이제 컨테이너의 XML 파일에 정의했으나 componentName 변수를 지정하지 않은 빈이 있다면, 이 빈은 DefaultComponentNamePostProcessor 클래스에서 먼저 처리해 기본 설정 값으로 componentName 변수 값을 지정하게 된다.

또한 후처리기를 사용하려면 스프링에도 사용한다는 사실을 알려야 한다. 즉, 빈 팩토리 컨테이너라면 후처리기 인스턴스를 지정하는 데 필요한 빈 팩토리 컨테이너의 addBeanPostProcessor 메서드를 호출해야 한다. 하지만 애플리케이션 컨텍스트 컨테이너는 컨테이너가 자동으로 클래스를 읽은 다음에 후처리기인지의 여부를 판단하므로 XML 파일에 정의하는 것만으로 충분하다.

빈에서 사용하는 init/destroy-method 속성을 각각의 빈마다 적용하는 것처럼 BeanPostProcessor 인터페이스는 컨테이너마다 적용한다는 점을 유념하자.

4.5 부모-자식 빈 정의

스프링에서는 추상 클래스와 구현 클래스를 XML 파일에 선언하는 일이 가능하다. 예를 들어 AbstractKlass가 추상 클래스고 Klass가 자식 클래스라면 다음처럼 빈을 정의할 수 있다.

```
<bean name="abstractKlass"
      class="com.madhusudhan.jscore.fundamentals.inherit.AbstractKlass"
      abstract="true" />
<bean name="klass"
      class="com.madhusudhan.jscore.fundamentals.inherit.Klass"
      parent="abstractKlass" />
```

추상 클래스로 선언한다면 볼드체로 표시한 부분처럼 해당 클래스의 abstract 속성을 true로 지정해야 한다. 그리고 자식 클래스는 parent 속성에 부모 클래스인 추상 클래스를 참조해야 한다.

4.6 요약

4장까지 살펴보면서 스프링의 핵심을 배우는 여행을 마쳤다.

지금까지 컨테이너의 기본 요소를 설명했으며 그 사용 방법도 알아보았다. 또한 빈에 프로퍼티를 지정하지 않아도 되도록 오토와이어링 빈의 사용에 관해서도 이야기했다. 빈 인스턴스화를 할 수 있는 여러 가지 방법, 정적 메서드나 팩토리 메서드를 이용하는 방법도 설명했으며 스프링이 처리하는 이벤트도 살펴봤다.

5장부터 살펴볼 스프링의 중요한 측면 하나는 스프링 JMS나 데이터베이스와 같은 엔터프라이즈 기능을 지원한다는 점이다. 5장에서는 스프링이 자바 메시징에 제공하는 편리함을 설명할 예정이다.

5 | 스프링 JMS

자바 메시징 서비스Java Message Service, JMS API는 10년 전쯤에 선을 보였으며 메시지 분산과 엔터프라이즈 애플리케이션 분야에서 큰 인기를 얻었다. 쉽고 표준화된 메시지 미들웨어 통합에 대한 수요가 이러한 기술을 만든 원동력이 되었다. JMS API는 간단하고 비교적 쉬운 편이지만 스프링은 이보다 한 걸음 더 나아가 조금 더 쉬운 프레임워크를 만들었다. 5장에서는 JMS의 기본을 소개하고 스프링이 JMS를 이용하여 어떻게 개발자의 삶을 편하게 했는지를 설명하겠다.

5.1 JMS 간단 소개

JMS는 결합하지 않은 애플리케이션 사이의 의사소통 메커니즘을 제공한다. 즉, 애플리케이션은 JMS 공급자를 통해 서로를 전혀 알지 못해도 의사소통을 하는 일이 가능해졌다.

예를 들면 PriceQuote라는 서비스는 가격 견적을 받고 싶어하는 누군가가 있는 채널로 가격 견적을 보낸다. 그런데 사실 PriceQuote 서비스 입장에서는 이 견적을 이용할 소비자가 누구인지는 전혀 알지 못한다.

이와 비슷하게 소비자 입장에서도 데이터 소스나 생산자에 대해서 전혀 아는 바가 없다. 즉, 그들 모두는 벽 뒤에 숨어있고 목적지destination나 채널을 사용해서만 교류를 하는 것이다. 생산자와 소비자는 원하는 메시지를 전달하고 받을 수만 있다면 서로의 존재는 전혀 상관하지 않는다.

JMS에는 세 가지 중요한 요소로 서비스 제공자, 생산자 그리고 소비자가 있다. 서비스 제공자가 중심에 있으며 다른 두 요소는 그저 데이터를 제공하는 사람과 그를 소비하는 사람일 뿐이다. 이때 아키텍처는 집중형 방식을 따르고 있다. 즉, JMS 서

버는 서비스 제공자를 바퀴축처럼 중심에 있도록 관리하며 생산자와 소비자는 바퀴처럼 움직이는 것이다.

이 아키텍처 운영의 전반적인 책임은 서비스 제공자가 진다. 서비스 제공자는 소비자가 메시지를 잘 받았는지 확인하는 동시에 생산자가 메시지를 보낼 수 있는지를 관리하게 된다. 지금까지 설명한 것보다 JMS를 더 자세히 알고 싶다면 이와 관련된 책을 찾아서 읽어보기 바란다.

5.1.1 메시징 모델

JMS에는 크게 두 가지 메시징 모델이 있다. 한 개는 포인트 투 포인트(P2P라고 부르기도 한다) 모델이며, 다른 하나는 발신/구독(Pub/Sub) 모델이다. 또한 각 모델을 언제 사용해야 하는지 지정한 이용 사례가 있다. 비록 이 두 모델은 메시지 발신과 수신 메커니즘이 근본적으로 다르지만 이 모델에 접근할 때는 단일화한 API, 즉 JMS를 사용하게 된다.

P2P 메시징

P2P 모델에서 메시지는 목적지를 거쳐서 소비자 한 명에게만 전달된다. 즉, 특정 소비자를 위한 것이다. 발신자는 메시지를 목적지로 보내고 소비자는 목적지에서 메시지를 받는다. 이는 마치 여러분 한 사람만을 위한 생일 축하 카드와 유사하다.

즉, P2P 모델에서 메시지를 큐Queue로 보내면 소비자 한 명만이 메시지를 받을 수 있는 것이다. 해당 큐에 몇백 명의 소비자가 연결되더라도 여전히 소비자 한 명만이 메시지를 전달받을 수 있다. 그게 누가 될지는 전혀 알 수가 없다. 간단히 말해 P2P 모델의 목적지는 큐다.

메시지를 목적지로 보내면 각각의 메시지를 복사한 사본을 받는 여러 구독자가 있을 수 있다. 즉, 발신자는 메시지를 단 한 번만 보내지만 해당 메시지에 관심 있는 발신자들은 이를 이용하려고 같은 목적지를 향한다는 뜻이다. 조금 전에 설명했듯이 JMS 공급자는 각 구독자가 메시지 사본을 잘 받았는지를 확인한다. 이는 마치 자동차 딜러가 독자 여러분과 그 외 수천 명에게 같은 제안서를 보내는 일과 같은 이치다. 이러한 발신/구독 메시징 모델의 목적지를 토픽Topic이라고 부른다.

5.2 스프링 JMS

스프링은 JMS API를 조금 더 단순화시켰다. 그러므로 스프링 JMS 사용법을 이해하기 위해 따라야 할 간단한 예시 소프트웨어 패턴이 있다. 먼저 소스 코드 예제를 살펴보기 전에 개괄적인 클래스와 클래스의 사용 방법을 설명하겠다.

스프링 JMS의 핵심은 JmsTemplate 클래스 하나가 중심이라는 점이다. 즉, 스프링 JMS에서는 이 클래스만 이해하면 스프링 JMS에 대한 실마리를 거의 잡은 것이라고 볼 수 있다. JmsTemplate 클래스는 메시지 이용하거나 생성하는 모두에 광범위하게 사용하지만 JmsTemplate 클래스를 사용하지 않을 때는 비동기식 메시지에서는 사용하지 않는다. 이는 이번 장 뒷부분에서 설명하겠다.

이제 JmsTemplate 클래스를 살펴보자.

5.2.1 모든 것의 모태가 되는 JmsTemplate 클래스

JmsTemplate 클래스는 메시지를 보낼 때와 받을 때 모두에 사용할 수 있다. 따라서 템플릿 클래스는 공급자에게 연결할 때 필요한 기반을 숨기며, 메시지를 발신/수신할 때 필요한 반복적인 인용 구문을 모두 제거했다. 또한 Connection 인터페이스와 같은 리소스 등을 보이지 않는 곳에서 관리한다.

스프링 JMS를 사용하려면 JmsTemplate 클래스의 인스턴스를 설정하고(일반 빈과 비슷하게), 이를 빈에 주입하는 것이 일반적인 방법이다. 여기서 우선순위 같은 서비스 품질Qualitiy of Service, QoS 프로퍼티나 클래스의 메시지 전달 방식을 클래스에 지정하는 것도 가능하지만, 이 중 ConnectionFactory 인터페이스의 객체는 프로퍼티 형태로 반드시 지정해야 한다는 점을 기억하자(Connection 프로퍼티는 클래스에서 연결해야 하는 서비스 제공자 정보를 제공하는 클래스다). 이 외의 다른 기능들을 위해서 사용하는 프로퍼티들(예를 들면 defaultDestination과 receiveTimeout 매개변수)에 대해서도 곧 살펴볼 것이다.

템플릿 클래스는 콜백 인터페이스를 사용해 동작한다. 즉, 메시지를 생성할 때는 MessageCreator, Session을 연결할 때는 SessionCallback, 메시지 생산자를 만들 때는 ProducerCallback과 같은 콜백 인터페이스를 사용한다. 한번 설정하거나 인스턴스화하면 JmsTemplate 클래스의 인스턴스는 스레드로부터 안전한thread-safe 인스턴스로 세션 손상의 우려 없이 여러 클래스에 공유할 수 있다.

JmsTemplate 클래스를 인스턴스화하거나 생성할 때 주로 사용하는 두 가지 방법이 있는데, 하나는 new 연산자를 사용해서 클래스의 인스턴스를 생성하는 것이고 다른 하나는 JmsTemplate 클래스를 XML 파일에 정의하는 것이다.

이때 두 가지 방법 모두 프로퍼티로 선언한 ConnectionFactory 인터페이스의 인스턴스를 인자로 전달할 필요가 있다.

다음 TradePublisher 클래스는 메시지 발행을 위해 JmsTemplate 클래스를 사용했다. JmsTemplate 클래스는 connectionFactory 객체를 생성자 인자로 전달하면서 인스턴스화했다.

```
public class TradePublisher {
  private String destinationName = "justspring.core.jms.trades";
```

```
    private JmsTemplate jmsTemplate = null;

    public void setConnectionFactory(ConnectionFactory connectionFactory) {
      // 주입한 ConnectionFactory 프로퍼티로
      // connectionFactory 객체를 생성함
      jmsTemplate = new JmsTemplate(connectionFactory);
    }
    ...
  }
```

TradePublisher 클래스를 JmsTempate 클래스의 인스턴스와 함께 생성했다면, 다음은 이 인스턴스를 메시지 발신에 사용하면 된다. jmsTemplate 인스턴스에는 메시지 발신 시 호출할 수 있는 send 메서드가 있는데 이 종류는 두 가지로, 하나는 목적지를 가진 것 또 하나는 이를 갖지 않은 것이다. 하지만 두 메서드 모두 MessageCreator 콜백 인터페이스를 반드시 인자로 가져야 한다. 이는 조금 이따가 알아보자. 다음은 이 인스턴스를 사용하는 예제다.

```
  public class TradePublisher {
    // 메시지 발신을 위해 jmsTemplate에 접근함
    public void publishTrade(final Trade t) {
      jmsTemplate.send(destinationName, new MessageCreator() {
        @Override
        public Message createMessage(Session session) throws JMSException {
          ObjectMessage msg = session.createObjectMessage();
          msg.setObject(t);
          return msg;
        }
      });
    }
  }
```

이 클래스의 요구 사항은 오직 ConnectionFactory 인터페이스를 프로퍼티 형태로 선언해 XML 파일에 기록한 발신자와 연결해서 빈을 생성할 때 주입할 수 있도록 하는 것이다. 물론 connectionFactory 빈을 사용해 JmsTemplate 클래스 객체를 인스턴스화해야만 한다.

다음 XML 파일을 살펴보자.

```
<bean name="tradePublisher"
      class="com.madhusudhan.jscore.jms.TradePublisher">
  <property name="connectionFactory" ref="connectionFactory" />
</bean>

<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
```

이전에 말했듯이 앞서 설명한 방법 이외에도 빈에 JmsTemplate 클래스를 사용할 수 있는 방법이 하나 더 있다. 템플릿 클래스를 연결 팩토리와 연결해 이를 빈에 주입하는 것이다.

이 모든 것은 XML 파일 안에서 해결할 수 있다. 잘 살펴보면 여기서 했던 일은 템플릿 클래스를 생성하는 역할을 애플리케이션에서 프레임워크로 옮긴 것이란 걸 알 수 있다.

```
<bean name="tradePublisher"
      class="com.madhusudhan.jscore.jms.TradePublisher">
  <property name="jmsTemplate" ref="jmsTemplate" />
</bean>
```

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
</bean>

<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
```

또한 빈은 setter 메서드를 통해 jmsTemplate 변수 값을 참조 값으로 갖게 될 것이다. TradePublisher 클래스는 다음 소스 코드처럼 템플릿과 연결되어 있다.

```
public class TradePublisher {
  private String destinationName = "justspring.core.jms.trades";
  private JmsTemplate jmsTemplate = null;

  public void setJmsTemplate(JmsTemplate jmsTemplate) {
    this.jmsTemplate = jmsTemplate;
  }

  public JmsTemplate getJmsTemplate() {
    return jmsTemplate;
  }

  public void publishTrade(final Trade t) {
    ...
  }
}
```

JmsTemplate 클래스는 JMS 서버로부터 메시지를 보내거나 받아야 할 때 관련 클래스로 주입될 것이고, 위 예제에서는 TradePublisher 클래스로 주입될 것이다.

위 XML 파일에서 분명하게 볼 수 있듯이 connectionFactory 빈으로 생성하는 객체는 이미 JmsTemplate 클래스와 명시적으로 연결되어 있다.

앞서 언급한 메시지를 보낼 때는 JmsTemplate 클래스에 속한 다음 세 개의 메서드를 사용한다.

```
send(MessageCreator msgCreator) // 기본 목적지
send(String destinationName, MessageCreator msgCreator)
send(Destination destinationName, MessageCreator msgCreator)
```

첫 번째 메서드는 메시지를 기본 설정한 목적지로 보낼 것이며, 나머지 두 개의 메서드는 목적지를 지정한다. MessageCreator 인자는 JMS 메시지를 생성하는 데 사용되는 콜백 인터페이스로, 오버라이딩되어야 할 createMessage(Session session) 메서드 하나를 가졌다. 이 세션 객체를 사용해 데이터가 있는 메시지 타입 하나를 생성할 수 있다.

좀 더 명확하게 알려면 TradePublisher 클래스에 구현한 publishTrade 메서드를 확인해보자. JMS에서는 메시지를 보내거나 받기 전에 모든 메시지가 미리 지정한 다섯 가지 타입인 TextMessage, BytesMessage, ObjectMessage, StreamMessage, MapMessage 중 하나로 반드시 선언해야 한다. 메시지 타입을 좀 더 자세히 알려면 JMS API를 따로 읽어보자.

5.2.2 메시지 발행하기

이번에는 메시지 발신 메커니즘을 이해할 수 있는 예제 하나를 살펴보도록 하자. StudentEnroller 클래스는 학생이 수업을 등록했을 때 호출하는 발행 컴포넌트다. 이 클래스의 목적은 등록한 학생을 무사히 JMS 목적지로 보내 이와 연관 있는 다른 클래스가 이를 받거나 행동할 수 있게 하는 것이다(예를 들어 과목 하나를 선

택하도록 학생을 부추기거나 도서관 설비나 카페, 연합 등의 정보를 제공하는 등의 경우다). 다음 소스 코드를 확인해보자.

```
public class StudentEnroller {
  private String destinationName = null;
  private JmsTemplate jmsTemplate = null;

  public void publish(final Student s) {
    getJmsTemplate().send(getDestinationName(), new MessageCreator() {
      @Override
      public Message createMessage(Session session) throws JMSException {
        TextMessage m = session.createTextMessage();
        m.setText("Enrolled Student: " + s.toString());
        return m;
      }
    });
  }
  // jmsTemplate와 destinationName 변수를 위한 setter, getter 메서드를 구현
  ...
}
```

소스 코드를 살펴보면 jmsTemplate 변수와 destinationName 변수가 있음을 알 수 있다. jmsTemplate 변수는 메시지를 보내는 데 사용하고, destinationName 변수는 어디로 메시지를 보내야 할지 그 장소(큐 혹은 토픽)를 정의해준다.

jmsTemplate와 destinationName 변수 값은 런타임 시에 StudentEnroller 클래스로 주입되고 StudentEnroller 클래스의 인스턴스가 보내는 메서드는 곧 설명할 StudentEnrollerClient 클래스에서 메시지를 보내고자 할 때 호출한다.

새로운 JMS 메시지를 생성하려면 스프링에서 제공하는 MessageCreator 인터페이스의 구현 클래스를 사용하고, 템플릿 클래스에 createMessage라는 콜백 메서드를 구현한 새로운 인스턴스를 생성해야 한다.

```
public Message createMessage(Session session) throws JMSException {
  TextMessage m = session.createTextMessage();
  m.setText("Enrolled Student: " + s.toString());
  return m;
}
```

여기서 적절한 메시지는 Session 인터페이스를 이용해서 생성하면 되는데, 이 예제에서 해당 메시지는 TextMessage 인터페이스를 말한다. 또한 빈을 이용해서 JmsTemplate 클래스를 StudentEnroller 클래스로 주입하기 전에 XML 파일에 관련 내용을 완전히 설정해야 한다. 템플릿 클래스에는 정의하고 참조해야 하는 connectionFactory 변수가 있어야 한다.

connectionFactory 변수는 각 JMS 공급자에 한정해 사용하며 앞으로 예제에서는 ActiveMQ를 JMS 공급자로 사용할 예정이다. 하지만 실제 소스 코드를 작성할 때는 ActiveMQ 외에 JMS와 호환되는 다른 공급자를 사용해도 상관없다. 즉, 프로그램은 어떤 공급자를 사용하든지 거기에 대응해서 문제 없이 작동해야 한다.

따라서 ActiveMQConnectionFactory 클래스를 사용하려면 brokerUrl 프로퍼티를 지정해야 한다.

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
</bean>

<bean id="connectionFactory"
```

```
        class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
  </bean>
```

마지막으로 할 작업은 XML 파일에 StudentEnroller 빈 자체를 jmsTemplate, destinationName 프로퍼티와 함께 선언하는 일이다.

```
  <bean name="studentEnroller"
        class="com.madhusudhan.jscore.jms.pub.StudentEnroller">
    <property name="destinationName" value="topic.STUDENTS" />
    <property name="jmsTemplate" ref="jmsTemplate" />
  </bean>
```

그럼 위 예제인 jms-pub-beans.xml 파일을 로드해 스프링 컨테이너를 생성하는 클라이언트인 StudentEnrollerClient 클래스를 살펴보자.

```
public class StudentEnrollerClient {
  private ApplicationContext ctx = null;
  private StudentEnroller enroller = null;
  public StudentEnrollerClient() {
    ctx = new ClassPathXmlApplicationContext("jms-pub-beans.xml");
    enroller = (StudentEnroller) ctx.getBean("studentEnroller");
  }
  public void publishStudent(Student s) {
    System.out.println("Publishing Student..");
    enroller.publish(s);
  }
  public static void main(String[] args) {
    StudentEnrollerClient client = new StudentEnrollerClient();
    Student s = new Student();
```

```
        client.publishStudent(s);
    }
}
```

위 예제에서 StudentEnrollerClient 클래스는 Student 클래스 객체 s와 함께 새롭게 발행 메서드를 호출하기 위해 enroller 변수를 선언했다. enroller 변수에는 이미 jmsTemplate와 destinationName 프로퍼티와의 의존성이 주입된 상태다(앞 XML 파일의 studentEnroller 빈 정의를 확인하자). 이제 ActiveMQ 서버를 실행시키면 98쪽 XML 파일에 정의했듯이 로컬 컴퓨터(localhost)의 포트 61616에서 작동할 것이다.

이제 StudentEnrollerClient 클래스를 실행시켜 모든 것이 완벽하게 적용되었는지 확인해보자. ActiveMQ 목적지에서 다음 메시지를 확인할 수 있어야 한다.

```
Publishing Student..
Successfully published student message to topic.STUDENTS
```

ActiveMQ는 요청한 목적지(이때는 topic.STUDENTS)가 존재하지 않는다면 목적지를 강제로 생성한다는 점을 기억하자.

5.2.3 기본 설정 목적지로 메시지 전송하기

기본 설정 목적지로 메시지를 전송하기 위한 경우라면, JmsTemplate 클래스의 send(MessageCreator msgCreator) 메서드를 이용하면 된다.

하지만 이 메서드를 XML 파일에 연결하는 작업은 필요하다. 메서드를 살펴보면 알 수 있듯이 여기서는 목적지에 대한 매개변수를 취하지 않는다. 즉, 메시지는 기본으로 지정된 목적지로 향하게 되지만 이 기본 설정 목적지는 XML 파일과 연결되어 있어야만 한다.

XML 파일에 목적지를 생성하려면, 보통 JMS의 서비스 제공자가 주는 javax.jms.Destination 클래스의 구현물을 사용해야 한다. ActiveMQ라면 빈으로 선언한 org.apache.activemq.command.ActiveMQTopic가 바로 목적지다.

```
<bean id="defaultDestination"
      class="org.apache.activemq.command.ActiveMQTopic">
  <constructor-arg value="topic.DEFAULT_STUDENTS" />
</bean>
```

그럼 이제 defaultDestination 프로퍼티를 jmsTemplate 빈에 추가해 위 목적지를 참조하게 하자.

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="defaultDestination" ref="defaultDestination" />
</bean>
```

이제 모든 소스 코드가 완성됐으니 발신에 관한 소스 코드를 확인해보자. 아래 소스 코드에서 send 메서드는 목적지로의 참조 값을 갖지 않았음을 알 수 있다.

```
public void publishToDefautDestination(final Student s) {
  getDefaultDestinationJmsTemplate().send(new MessageCreator() {
    @Override
    public Message createMessage(Session session) throws JMSException {
      ....
    }
  });
}
```

5.2.4 토픽 선언

JmsTemplate 클래스는 기본 메시징 모델이 P2P 모델이라고 생각하므로 목적지가 큐라고 생각한다. 하지만 발행/구독 형식으로 바꾸고 싶다면 pubSubDomain라는 프로퍼티를 jmsTemplate에 연결하면 된다.

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  ...
  <property name="pubSubDomain" value="true" />
</bean>
```

이렇게 하면 메시지를 토픽으로 발행할 것이다. 다음처럼 설정 파일에 큐나 토픽을 생성하는 것을 잊지 말자(이 경우는 ActiveMQ).

```
<!-- 큐 -->
<bean id="defaultDestination"
      class="org.apache.activemq.command.ActiveMQQueue">
  <constructor-arg value="queue.STUDENTS" />
</bean>

<!-- 토픽 -->
<bean id="defaultDestination"
      class="org.apache.activemq.command.ActiveMQTopic">
  <constructor-arg value="topic.STUDENTS" />
</bean>
```

5.2.5 메시지 수신하기

JmsTemplate 클래스를 사용하게 되면 메시지를 받는 일이 매우 수월해지긴 하지만 수신 방식을 고려하면 조금 복잡한 문제가 도사리고 있다. 이 경우 메시지를 수신하는 두 가지 방식으로 동기 방식과 비동기 방식을 고려해야 한다.

동기 방식을 이용할 경우 서버로부터 적어도 메시지 하나를 수신할 때까지 다른 활동은 아무것도 할 수 없다. 즉, 스레드가 응답 없는 수신 메서드를 호출했다면 메시지를 가져갈 때까지 무기한으로 기다린다.

이는 싱글 스레드와 같은 동작 방식이므로 개인적인 의견으로는 어떤 특정한 혹은 강력한 이유가 있지 않고서는 이 방식은 이용하지 않는 것이 좋다. 만약 동기적 수신 메서드 외에 다른 방법이 없는 경우에는 timeout 값을 설정해 스레드가 해당 시간 이후 종료할 수 있게 한 후에 이용하기 바란다.

비동기 방식을 이용할 경우 클라이언트가 어떤 목적지의 메시지 수신에 관심을 보이는지를 서비스 제공자에게 알려준다. 해당 목적지에 메시지가 도착하게 되면 서비스 제공자는 이 메시지에 관심을 보였던 클라이언트를 확인하여 메시지를 적절한 위치에 보내게 된다.

이러한 방식으로 메시지를 받기 위해 JmsTemplate 클래스의 receive 메서드를 사용한다.

5.2.6 동기 메시지 수신하기

JmsTemplate 클래스의 receive 메서드는 목적지를 인자로 얻으면 그곳에서부터 메시지를 받기 시작한다. JMS의 서비스 제공자로 접속해 메시지를 수신하는 JobTaskReceiver 클래스를 확인해보자(여기서는 JmsTemplate 클래스 객체를 인스턴스에 주입한다).

```
public class JobsReceiver {
  private JmsTemplate jmsTemplate = null;
  private String destinationName = null;

  public void receiveMessages() {
    Message msg = getJmsTemplate().receive(destinationName);
```

```
        System.out.println("Job Received: " + msg);
      }
      ...
    }
```

이 클래스에서 receive 메서드는 목적지가 메시지를 한 개라도 가질 때까지 계속해서 기다린다. 이전에 설명했듯이 동기 메시지 수신(receive 메서드 사용)은 큐에 메시지가 없을 때도 CPU 사이클을 낭비할 수 있는 블로킹 콜blocking call이다. 그러므로 receiveTimeout 프로퍼티에 적절한 값을 넣는 것이 좋다.

receiveTimeout 프로퍼티는 JmsTemplate 클래스의 속성 중 하나로 다음처럼 설정 파일에 선언해야 한다.

```
    <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
      <property name="connectionFactory" ref="connectionFactory" />
      <property name="receiveTimeout" value="2000" />
      ...
    </bean>
```

위 코드에서 정의했듯이 JobTaskReceiver 클래스는 아무 메시지도 전달되지 않았을 경우 2초 후에 종료한다.

메시지를 defaultDestination 프로퍼티에서 수신하는 것도 가능하다. 이를 위해서는 이전에 구현했던 것처럼 defaultDestination 프로퍼티를 jmsTemplate 빈과 연결하면 된다. 간단하다!

```
    <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
      <property name="defaultDestination" value="topic.DEFAULT_JOBS" />
      ...
    </bean>
```

메시지를 비동기 수신하는 경우는 조금 다르다. 어떻게 하는지 살펴보자.

5.2.7 비동기 메시지 수신하기

메시지를 비동기 수신하려면 다음 두 가지만 기억하면 된다.

- MessageListener 인터페이스를 구현하는 클래스를 생성한다.
- XML에 정의한 스프링 JMS 컨테이너를 위 리스너의 참조 값과 연결한다. XXXContainer에 대해서는 조금 뒤에 이야기하도록 하자.

첫 번째로 비동기 클라이언트는 JMS API의 인터페이스인 MessageListener를 반드시 구현해야 한다. 그리고 이 인터페이스는 클래스에 구현되어야 하는 onMessage라는 메서드를 하나 가진다. 다음 예제의 BookOrderMessageListener 클래스는 MessageListener 인터페이스를 구현하는 간단한 클래스로 메시지를 콘솔에 출력하는 단순한 메서드 하나만 가진다.

```
public class BookOrderMessageListener implements MessageListener {
  @Override
  public void onMessage(Message msg) {
    System.out.println("Book order received:" + msg.toString());
  }
}
```

두 번째는 컨테이너를 연결하는 작업을 설명한다. 단, 여기서 설명하는 컨테이너는 ApplicationContext나 BeanFactory 인터페이스가 아님을 주의하자.

이 컨테이너는 메시지 수신 클라이언트에서 사용하려고 스프링에서 제공하는 유틸리티 클래스로, Connection 인터페이스와 Session 인터페이스의 복잡함을 숨기는 간단하지만 강력한 클래스다. 이는 JMS의 서비스 제공자로부터 데이터를 가져

오고 리스너로 보내주는 역할을 하게 되며, ConnectionFactory(즉, JMS의 서비스 제공자로 연결하는 통로) 인터페이스와 리스너 클래스로의 참조 값을 가진다.

이제 어떻게 동작하는지 자세히 살펴보기 위해 컨테이너와 리스너를 다음 소스 코드처럼 연결해보도록 하자. DefaultMessageListenerContainer 클래스의 인스턴스를 정의한 다음 connectionFactory, destination, messageListener 프로퍼티로 인스턴스를 주입하면 된다. 이때 messageListener 클래스가 리스너 클래스를 참조한다는 점을 유의하자.

```
<bean id="bookOrderListener"
      class="com.madhusudhan.jscore.jms.async.BookOrderMessageListener" />

<bean id="defaultListenerContainer" class=
 "org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="destination" ref="defaultDestination" />
  <property name="messageListener" ref="bookOrderListener" />
</bean>
```

위와 같은 연결이 끝나고 빈을 로드하기 위해 StudentEnrollerClient 클래스를 실행시키면 JMS 서버로부터 메시지 수신을 기다리는 messageListener 인스턴스부터 동작할 것이다. 목적지로 메시지를 보내면 StudentEnrollerClient 클래스에 messageListener가 보낸 해당 메시지가 나타나는 것을 확인할 수 있다.

5.2.8 스프링 메시지 컨테이너

'5.2.7 비동기 메시지 수신하기'에서 DefaultMessageListenerContainer 클래스를 사용하는 방법을 알아보았다. 스프링은 메시지를 비동기 수신하기 위해서 DefaultMessageListenerContainer 클래스를 포함한 세 가지 유형의 컨테이너 클래스를 제공하며, 다른 두 가지는 SimpleMessageListenerContainer와 ServerSessionMessageListenerContainer 클래스다.

SimpleMessageListenerContainer 클래스는 위 세 가지 유형 중 가장 간단하게 사용할 수 있지만 상업적 용도로는 적합하지 않다. 반면에 ServerSessionMessageListenerContainer 클래스는 DefaultMessageListenerContainer 클래스와 비교했을 때 기능적, 복합적 측면에서 우위에 있다. 이는 JMS 세션과 직접 연결해서 작업해야 할 때나 XA 트랜잭션이 필요한 상황에 사용하는 것이 좋다. DefaultMessageListenerContainer 클래스는 애플리케이션 대부분에 대응해 사용하기에 좋으며 외부 트랜잭션 접근이 용이하다. 위 세 가지 유형 중 애플리케이션의 요구 사항에 가장 부합하는 컨테이너 클래스를 선택해 사용하면 된다.

5.2.9 메시지 컨버터

메시지를 보내기 위해서 해야 할 일 하나는 도메인 객체를 미리 정의한 다섯 가지의 JMS 메시지 타입으로 변환하는 것이다.

Trade나 Order 같은 도메인 객체는 적절한 직렬화 자바 객체라고 하더라도 보내거나 받을 수 없다. 그러므로 도메인 객체를 보내려면 이를 적절한 JMS 메시지 타입으로 변환해야 한다. 다음 publish 메서드의 소스 코드를 확인해보자.

```
    public void publish(final Student s) {
      getJmsTemplate().send(getDestinationName(), new MessageCreator() {
        @Override
        public Message createMessage(Session session) throws JMSException {
```

```
            TextMessage m = session.createTextMessage();
            m.setText("Enrolled Student: " + s.toString());
            return m;
          }
        }
        ...
    }
```

이 클래스에서는 메시지를 보내기 위해 Session 인터페이스(적절한 객체를 생성하기 위해)와 MessageCreator 콜백 인터페이스(session 인터페이스의 객체를 제공해준다)로 작업해야 한다는 사실을 눈여겨보자. 이 때문에 소스 코드가 별로 깔끔해 보이지는 않는다.

Publish 메서드는 컨버터의 도움을 받아 다음처럼 좀 더 간단하고 명확하게 개선할 수 있다. 다음 소스 코드를 확인해보자.

```
    public void publish(final Student student) {
      jmsTemplate.convertAndSend(destinationName, student);
    }
```

convertAndSend 메서드는 스프링의 MessageConverter 인터페이스를 이용해 모든 추가 반복 코드를 없앨 수 있다. 즉, MessageConverter 인터페이스는 개발자들이 고생할 필요가 없도록 쉽게 변환을 해준다는 뜻이다. 일단 JmsTemplate 클래스에 컨버터를 연결하면 더 이상은 변환 작업을 걱정하지 않아도 된다. 물론 컨버터 하나를 만들 필요는 있다.

그럼 컨버터가 어떻게 동작하는지를 살펴보기 위해 우선 MessageConverter 인터페이스를 구현하는 클래스를 만들어보자. MessageConverter 인터페이스는 fromMessage와 toMesage라는 두 개의 메서드를 가지고 있는데, 이름에서 알 수

있듯이 JMS 메시지에서 도메인 객체로 변환하거나 그 반대로 사용할 수 있도록 이 메서드를 구현해야 한다.

다음 소스 코드는 Account 객체에 사용하는 기본적인 컨버터 클래스다.

```
public class AccountConveter implements MessageConverter {
  @Override
  public Object fromMessage(Message msg) throws JMSException,
                                          MessageConversionException {
    Account t = (Account) ((ObjectMessage) msg).getObject();
    System.out.println("fromMessage: " + msg.toString());
    return t;
  }

  @Override
  public Message toMessage(Object obj, Session session) throws
                          JMSException, MessageConversionException {
    ObjectMessage objMsg = session.createObjectMessage();
    objMsg.setObject((Account) obj);
    System.out.println("toMessage: " + objMsg.toString());
    return objMsg;
  }
}
```

fromMessage 메서드에서 JMS의 msg 객체가 t 객체를 가져와 이를 실제 도메인 객체로 변환했다. 반면에 toMessage 메서드는 전달한 session 객체를 사용해 objMsg 객체를 생성하고 setObject 메서드를 사용해서 도메인 객체를 넣어 주었다. 이제 AccountConveter 클래스를 완성했다면 다음으로 해야 할 일은 jmsTemplate 빈으로 JmsTemplate 클래스의 객체와 연결하는 일이다.

```
<bean id="accountConverter"
      class="com.madhusudhan.jscore.jms.convert.AccountConverter" />

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="messageConverter" ref="accountConverter" />
  ....
</bean>
```

발행자와 수신자 소스 코드에서는 send와 receive 메서드를 convertAndSend와 receiveAndConvert 메서드로 변경하여 수신/발신 시 컨버터 빈을 사용하도록 만들자.

```
// 발행자
public void publish(final Student student) {
  jmsTemplate.convertAndSend(destinationName, student);
}

// 수신자
public void receive() {
   ObjectMessag e o = jmsTemplate.receiveAndSend(destinationName);
}
```

이제 t 객체로 메시지를 보낼 때마다 jmsTemplate 빈은 t 객체를 objMsg 객체로 변환하려고 컨버터를 사용할 것이고, 메시지를 수신할 때도 jmsTemplate 빈은 똑같은 컨버터를 사용할 것이다. 즉, 컨버터를 한번 구현하면 언제 어디서든 사용할 수 있다는 뜻이다.

5.3 요약

5장에서는 자바 메시징의 동작에 관해 이야기했다. JMS에 관한 내용과 스프링의 JmsTemplate 클래스를 사용하는 방법까지 간단히 알아보았고, 템플릿 클래스를 사용해서 메시지를 발행하는 방법과 메시지 컨테이너라고 불리는 스프링의 클래스를 사용해서 메시지를 동기, 비동기 수신하는 방법도 알아보았다. 마지막으로는 JMS 메시지 타입을 비즈니스 도메인 타입으로 변경하고 반복적으로 인용하는 구문을 삭제할 수 있는 컨버터도 다루었다.

6장에서는 스프링의 JDBC와 하이버네이트을 사용한 데이터의 퍼시스턴스와 검색을 설명하도록 하겠다.

6 | 스프링 데이터

데이터 저장과 검색 작업은 엔터프라이즈 업계에서는 필수적인 작업이다. 이런 작업을 도와주는 JDBC는 다양한 데이터베이스에 매우 쉽고 편안하게 접근할 수 있도록 만들어진 도구로 짧은 기간에 인기를 얻게 되었는데, 이는 MySQL, Oracle, Sybase 등 다양한 관계형 데이터베이스 시스템에 단일 API로 접근할 수 있다는 이유 때문이다.

스프링은 한 걸음 더 나아가 JDBC를 추상화하여 훨씬 가벼운 프레임워크를 만들었다. 즉, JDBC에 스프링을 곁들여 사용함으로써 데이터베이스에 접근하는 훨씬 더 좋은 방법을 제시한 것이다. 하지만 이 방법에도 한계가 있다. 객체 관계형 매핑Object Relational Mapping, ORM을 완전히 지원하지는 않기 때문이다. 우리는 여전히 SQL을 통해 데이터에 접근해야 한다. 하이버네이트는 이러한 상황을 틈타 개발했으며 매우 높은 인기를 얻게 되었다.

하이버네이트는 이미 몇 백만 번 이상 다운로드되었으며 현재 매우 강력하고 인기 있는 오픈 소스 프레임워크로 사용되고 있다. 스프링은 이렇듯 이미 강력한 기능을 가진 하이버네이트에 추상화를 추가함으로써 이를 더 개선했다.

6장에서는 데이터베이스에 접근할 때 연결성이나 구문 문제를 우려할 필요가 없도록 스프링을 어떻게 효과적으로 사용할지를 알아볼 것이다. 그런 다음 하이버네이트을 이용한 스프링의 ORM 지원도 살펴볼 것이다.

6.1 JDBC와 하이버네이트

스프링과 JDBC의 결합은 객체와 관계형 데이터 사이의 격차를 해결하지는 못했다. 하지만 JDBC는 데이터베이스 관련 작업을 할 때 자바 개발자가 여전히 우선으로 선택하는 인터페이스 중 하나로, 다양한 데이터베이스에 접근할 때 발생하는 복잡성을 없애는 명확하고 간단한 API를 제공한다.

JDBC 관련 작업을 하는 많은 개발자는 여전히 단 한 줄의 데이터만 가져오려 해도 많은 중복 소스 코드나 상용 구문을 작성(복사/붙여넣기까지!)해야 한다고 불평한다. 스프링은 이러한 부분을 과감하게 개선할 수 있도록 만들었다. 간단한 템플릿 패턴을 사용해 데이터베이스 접근을 혁신했고 모든 반복 구문을 스프링에 내장했다.

덕분에 필요 없는 부트스트래핑이나 리소스 관리 혹은 짜증나는 예외 상황을 더는 신경 쓰지 않아도 된다. 인제야 여러분이 사용하려던 기술을 사용할 수 있게 되었다는 뜻이다. 즉, 오직 비즈니스 로직만 작성하면 되는 것이다. 이제 스프링이 어떻게 이러한 동작을 할 수 있게 되었는지 알아보도록 하자.

두 번째로 고려해야 할 부분은 개발자가 관계형 개체를 소스 코드 안의 객체인 것처럼 작업하길 원한다는 점이다. 관계형 개체는 자바 객체와는 다르므로 그 사이에 브릿지가 없다면 이러한 개발자들의 소원을 이루어주긴 어려운 것이 현실이다. 하지만 이러한 차이를 해결할 수 있도록 만들어진 소프트웨어를 간단히 객체 관계형 매핑 프레임워크라고 하며 하이버네이트, JDO(Java Data Objects), 마이바티스(MyBatis [01]), 톱링크(TopLink)가 이 범주에 포함된다. ORM 도구를 사용하면 JDBC에서처럼 낮은 레벨에서의 작업이 필요 없으며 데이터를 객체로 다룰 수 있게 된다.

01 원서에는 아이바티스(iBatis)라고 표기되었으나 마이바티스로 명칭이 변경되어 수정했습니다.

예를 들어 많은 행(Movie)을 가진 MOVIES라는 테이블이 있다고 가정하자. 프로그램에서는 이 관계형 개체와 비슷한 타입으로 Movie 객체로 모델링하고, 보이지 않는 곳에서 프레임워크가 MOVIE의 행 하나를 Movie 도메인 객체로 매핑할 것이다.

6.1.1 스프링 JDBC

JDBC가 데이터 접근을 위한 간단한 API를 제공하고 있음에는 동의하지만 막상 코딩할 때가 되면 초보적인 소스 코드만을 작성해야 하므로 이를 사용하는 일은 여전히 매우 번거롭다. 어떤 사람들은 소스 코드 중 80%나 되는 양이 반복해서 사용하는 부분이라고까지 이야기하는데, 이는 소스 코드 재사용이 일반화된 요즘에는 있을 수 없는 일이다.

이를 개선하기 위해 스프링이 다음과 같은 일을 해냈다. 애플리케이션의 사용 목적에만 집중할 수 있도록 모든 리소스 관리를 떼어낸 것이다. 사실 스프링이 템플릿 설계 패턴을 '재사용'함으로써 데이터베이스와의 통신을 매우 쉽고 깔끔하게 할 수 있도록 만든다는 점은 그리 놀라운 일은 아니다. JDBC 패키지는 JdbcTemplate 클래스 하나만 핵심으로 하는데, 이 클래스는 컴포넌트에서 데이터에 접근하는 부분에서 가장 중요한 역할을 한다.

JdbcTemplate 클래스

앞서 설명한 것처럼 JdbcTemplate 클래스는 가장 중요하면서도 유용한 핵심 클래스다. 이 클래스로 작업의 대부분을 할 수 있지만 좀 더 세련된 소스 코드 작성을 위해서 JdbcTemplate의 두 가지 변형 클래스인 SimpleJdbcTemplate와 NamedParameterJdbcTemplate 클래스를 사용하는 편이 좋다

JdbcTemplate 클래스는 기존 소스 코드를 사용하거나, 표준 SQL 쿼리를 사용해서 테이블을 조회하거나, 저장 프로시저를 호출해 PrepraredStatement 인터페

이스로 데이터를 삽입하거나 업데이트하는 데이터베이스 작업의 일반적인 기능들을 제공한다. 또한 ResultSet 인터페이스의 데이터를 효율적이고 효과적으로 반복해서 사용하는 일도 가능하다.

연결 관리와 리소스 풀링, 예외 관리 부분은 사용자가 알 수 없도록 숨겨져 있으며 스프링의 RuntimeException 클래스가 데이터베이스 관련 예외 처리를 하므로 try-catch 문을 써서 예외를 찾으려고 소스 코드를 지저분하게 만들 필요가 없다.

템플릿 설계 패턴은 JdbTemplate 클래스에 콜백 인터페이스를 구현하는 데 필요한 비즈니스 로직을 생성하면 된다. 예를 들어 PreparedStatement 인터페이스의 객체를 생성할 때는 PreparedStatementCallback 인터페이스를 사용하며, 도메인 객체에 사용할 ResultSet 인터페이스의 객체는 RowCallbackHandler 인터페이스에서 가져온다. 또한 저장한 프로세스를 실행할 때는 CallableStatementCallback 인터페이스를 사용한다. 이제 이 콜백 인터페이스들을 알아보자.

JdbcTemplate 클래스 환경 설정하기

JdbcTemplate 클래스로 작업을 시작하기 전에 몇 가지 세부 사항을 확인할 필요가 있다.

첫 번째로 클래스가 데이터베이스 연결될 수 있도록 설정하기 위해 DataSource 인터페이스를 JdbcTemplate 클래스에 제공해야 한다. JMS를 설명한 이전 장에서 설명한 것처럼 ConnectionFactory 인터페이스가 JMS 제공자로 통하는 문이었다면 DataSource 인터페이스는 데이터베이스로 통하는 문으로 생각해도 무방하다. 즉, JdbcTemplate 클래스가 데이터베이스에 접속하는 데 사용하는 연결 정보를 제공한다.

DataSource 인터페이스는 다음 예제처럼 설정하면 된다. 여기서는 DataSource 인터페이스를 생성하기 위해 오픈 소스 JDBC 프레임워크인 Apache commons DBCP[Database Connection Pool]를 사용했다.

```
<bean id="movieDataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  <property name="driverClassName" value="${jdbc.driver}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

DataSource 인터페이스를 생성했다면 이번에는 jdbcTemplate 빈을 정의하자. 이때 크게 두 가지 선택 사항이 있다. 첫 번째는 데이터 접근 객체[Data Access Object, DAO] 안에서 DataSource 인터페이스의 빈을 JdbcTemplate 클래스의 빈에 주입함으로써 인스턴스화하는 방법이고, 두 번째는 JdbcTemplate 클래스의 빈을 XML 파일에 정의하고 DataSource 인터페이스를 연결한 다음 JdbcTemplate 클래스 빈이 참조하는 값을 XXXDAO 클래스의 빈에 주입하는 방법이다.

다음 소스 코드는 두 번째 방법을 사용한다.

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource" ref="movieDataSource" />
</bean>
<bean id="movieDataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  ....
</bean>
```

JdbcTemplate 클래스 객체는 스레드로부터 안전한 객체로, 한번 설정하면 몇 개의 XXXDAO 클래스 객체든 상관없이 주입할 수 있다.

MOVIES 데이터베이스에 접근하는 MovieDAO 클래스를 만들어보도록 하자.

```
public class MovieDAO implements IMovieDAO {
  private JdbcTemplate jdbcTemplate = null;

  private void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

  private JdbcTemplate getJdbcTemplate() {
    return this.jdbcTemplate;
  }
  ...
}
```

IMovieDAO는 movie 행과 관련 있는 모든 기능을 가진 인터페이스다.

```
public interface IMovieDAO {
  public Movie getMovie(String id);
  public String getStars(String title);
  public List<Movie> getMovies(String sql);
  public List<Movie> getAllMovies();
  public void insertMovie(Movie m);
  public void updateMovie(Movie m);
  public void deleteMovie(String id);
  public void deleteAllMovies();
}
```

MovieDAO 클래스는 jdbcTemplate을 멤버 변수로 가지고 있으며 이는 XML 파일의 데이터 소스와 연결하도록 설정해 구현 클래스(MovieDAO)에 주입한다.

연결하도록 설정한 XML 파일은 다음과 같다.

```
<bean id="movieDao" class="com.madhusudhan.jscore.data.MovieDAO">
  <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource" ref="movieDataSource" />
</bean>

<bean id="movieDataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
  ....
</bean>
```

이것이 끝이다! 이제 JdbcTemplate 클래스의 환경 설정은 완료했고 언제든지 사용할 수 있다. 이제 템플릿이 무엇을 할 수 있는지에 집중해보도록 하자.

JdbcTemplate 클래스 사용

movie star 값을 가져오는 가장 쉬운 방법은 movie title과 같은 조건을 사용하는 것이다. 또한 단순하게 모든 movie star 값을 가져오려면 'select stars from MOVIES'와 같은 간단한 SQL 쿼리문을 만들면 된다. MovieDAO 클래스는 이처럼 제목을 인자로 받아 콤마로 구분하는 stars 변수들의 목록을 반환하는 getStars 메서드를 가진다. 이를 구현한 다음 소스 코드를 확인해보자.

```
@Override
public String getStars(String title) {
  String stars = getJdbcTemplate().queryForObject(
    "select stars from MOVIES where title='" + title + "'", String.class
  );
  return stars;
}
```

쿼리는 JmsTemplate 클래스의 queryForObject 메서드를 사용해서 실행한다. 이 메서드는 두 개의 매개변수, 즉 변수를 묶지 않은 SQL 쿼리문과 결과 값을 예상하는 타입의 매개변수를 가진다.

여기서 예상하는 결과 값은 콤마로 구분한 starts 변수들의 목록이다. 쿼리문을 매개변수화하면 조금 더 개선한 형태가 되는데, 다음 소스 코드에서는 where절에 변수를 묶어 사용하면서 쿼리를 변형했다.

```
@Override
public String getStars(String title) {
  // where 절을 사용하자
  String stars = getJdbcTemplate().queryForObject(
    "select stars from MOVIES where title=?",
    new Object[]{title}, String.class
  );

  return stars;
}
```

여기서 알 수 있듯이 두 번째 인자는 메서드 인자로 전달하는 편이 좋다.

queryForInt, queryForList, queryForMap과 같은 queryForXXX의 다양한 메

서드를 JdbcTemplate 클래스에 정의했으니, 이를 확인하려면 스프링의 API를 참조하기 바란다.

도메인 객체 반환

바로 앞에서 설명한 소스 코드의 쿼리는 데이터 하나만을 반환한다. 이때는 movie star 값을 반환할 것이다.

그렇다면 쿼리의 id 값으로 movie 객체를 반환받고 싶을 경우에는 어떻게 해야 할까? JdbcTemplate 클래스의 queryForObject 메서드를 사용하면 편리하다. 여기에 RowMapper 인터페이스의 인스턴스를 추가로 전달하면 된다.

JDBC API는 rs 객체를 반환하므로 ResultSet 인터페이스 모든 열의 데이터를 도메인 객체에 매핑해야만 한다. 스프링은 RowMapper 인터페이스를 제공함으로써 이러한 반복적인 프로세스를 없앨 수 있다. 간단히 말해서 RowMapper 인터페이스는 테이블의 행 데이터를 도메인 객체로 매핑해주는 인터페이스다. 따라서 이 인터페이스의 구현 클래스는 mapRow 메서드를 구현해야만 한다. 여기서 우리가 해야 할 것은 테이블의 열 데이터를 movie 객체로 매핑하는 인터페이스의 구현 클래스를 작성하는 일이다.

movie 도메인 객체를 위해 이를 구현하자.

```
public class MovieRowMapper implements RowMapper {
  Movie movie = null;
  public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
    movie = new Movie();

    // ResultSet 인스턴스에서 데이터를 추출해 Movie 객체로 지정하는 곳
    movie.setID(rs.getString("id"));
    movie.setTitle(rs.getString("title"));
```

```
        ...
        return movie;
      }
    }
```

여기서 기본적으로 구현해야 하는 것은 관련 열 값을 rs 객체에서 추출해 movie 도메인 객체로 이동하고 이를 반환하는 일이다. 이제 MovieRowMapper 클래스를 구현했으니 값을 받아오는 JdbcTemplate 클래스의 인스턴스를 이용하자.

```
    @Override
    public Movie getMovie(String id) {
      String sql = "select * from MOVIES where id=?";
      return getJdbcTemplate().queryForObject(
        sql, new Object[]{id}, new MovieRowMapper()
      );
    }
```

JdbcTemplate 클래스는 인자를 묶고 쿼리가 반환한 ResultSet 인터페이스의 인스턴스와 함께 MovieRowMapper 클래스를 호출해 쿼리를 실행한다. 하지만 위 예제대로라면 해당 쿼리의 결과 값을 각각의 movie 객체로 얻게 된다. 이 대신 리스트 형식으로 모든 movie 객체를 결과 값으로 얻으려면 이 구조를 조금 더 개선할 필요가 있다.

MovieRowMapper 클래스를 모든 movie 객체를 반환할 때 사용해보자. 그러려면 다음처럼 RowMapperResultSetExtractor 클래스로 래핑해야 한다.

```
    public List getAllMovies() {
      RowMapper mapper = new MovieRowMapper();
      String sql = "select * from MOVIES";
```

```
    return getJdbcTemplate().query(sql,
    new RowMapperResultSetExtractor(mapper,10));
  }
```

데이터 처리

데이터를 삽입하거나 업데이트 또는 삭제하려면 jdbceTemplate.update 메서드를 사용하면 된다.

다음 소스 코드는 movie 객체를 데이터베이스로 삽입하는 예다.

```
  @Override
  public void insertMovie(Movie m) {
    String sql = "insert into MOVIES (ID, TITLE, GENRE, SYNOPSIS)
                  values(?,?,?,?)";

  // 매개변수 삽입
  Object[] params =
    new Object[] { m.getID(), m.getTitle(), m.getGenre(), m.getSynopsis() };
    // 매개변수 타입 삽입
    int[] types = new int[] { Types.VARCHAR, Types.VARCHAR,
                              Types.VARCHAR, Types.VARCHAR };
    // 쿼리 값 반환
    jdbcTemplate.update(sql, params, types);
  }
```

해당 메서드의 두 번째와 세 번째 매개변수는 입력 값과 각 데이터 타입을 나타낸다. 이와 비슷하게 movie 객체 값 하나를 데이터베이스에서 삭제하는 일은 쉽다.

```
  @Override
```

```
public void deleteMovie(String id) {
  String sql = "delete from MOVIES where ID=?";
  Object[] params = new Object[] { id };
  jdbcTemplate.update(sql, params);
}
```

모든 movie 객체를 삭제하려면 다음 소스 코드를 사용하면 된다.

```
@Override
public void deleteAllMovies() {
  String sql = "delete from MOVIES";
  jdbcTemplate.update(sql);
}
```

update 메서드를 사용하면 저장한 프로세스를 사용하는 일도 매우 쉽다.

```
public void deleteAllMovies() {
  String sql = "call MOVIES.DELETE_ALL_MOVIES";
  jdbcTemplate.update(sql);
}
```

지금까지 살펴봤듯이 JdbcTemplate 클래스는 데이터베이스에 접근하는 부담을 상당히 완화시켰다. 템플릿 클래스의 여러 유용한 메서드는 스프링의 API를 참조하기 바란다.

6.1.2 하이버네이트

하이버네이트는 XML 파일을 읽어서 데이터베이스의 열 값을 객체로 매핑하는 방식을 사용한다. 이를 위해서 XML 파일에 도메인 객체와 테이블의 열 데이터의 매핑 데이터를 정의해야 한다.

각 매핑 데이터가 담긴 XML 파일은 .hbm.xml 형태여야 한다. 스프링은 이를 한 단계 더 추상화하고 데이터베이스에 접근하려고 HibernateTemplate과 같은 클래스를 제공한다. 하지만 스프링에서는 이보다 하이버네이트의 API를 사용하는 편을 더 선호한다. 이는 잠시 후에 살펴보기로 하고 먼저 필요한 XML 파일을 어떻게 준비해야 하는지 알아보자.

다음은 하이버네이트 매핑 법칙을 사용해 MOVIE 객체를 정의해보았다.

```
<hibernate-mapping>
  <class name="com.madhusudhan.jscore.data.Movie" table="MOVIES">
    <id name="id" column="ID">
      <generator class="assigned" />
    </id>
    <property name="title" column="TITLE" />
    <property name="genre" column="GENRE" />
    <property name="synopsis" column="SYNOPSIS" />
  </class>
</hibernate-mapping>
```

class 속성은 실제 도메인 클래스, 이 예제에서는 Movie를 정의한다. 〈id /〉 요소는 프라이머리primary 키며 할당assign되어 있는데, 이 말은 프라이머리 키를 지정하는 일을 애플리케이션이 해야 한다는 뜻이다. 그 이외의 프로퍼티들은 MOVIES 테이블 안의 각 열 데이터와 매핑되어 있다.

또한 하이버네이트는 데이터베이스에 접근하려고 할 때 사용하기 위한 session 객체가 있어야 하는데, session 객체는 SessionFactory 인터페이스에서 생성한다. 스프링에서는 SessionFactory 인터페이스의 인스턴스를 생성하기 위해 스프링의 LocalSessionFactoryBean 클래스를 사용할 수 있으며, 연결해야 할 데이터 소스와 함께 하이버네이트 프로퍼티와 매핑할 리소스를 필요로 한다.

팩토리 빈의 hibernateProperties 프로퍼티는 데이터베이스의 dialect 옵션, 풀 pool 크기 옵션 등의 특정 프로퍼티를 사용할 수 있게 하고, mappingResources 프로퍼티는 매핑데이터가 담긴 XML 파일을 로드하게 한다(이 예제에서는 Movie.hbm.xml이다).

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource" ref="movieDataSource" />
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
        net.sf.hibernate.dialect.MySQLDialect
      </prop>
      <prop key="hibernate.show_sql">false</prop>
    </props>
  </property>
  <property name="mappingResources">
    <list>
      <value>Movie.hbm.xml</value>
    </list>
  </property>
</bean>
```

기존에 사용한 HibernateTemplate 클래스

sessionFactory 인터페이스의 빈을 정의했으니 다음으로 HibernateTemplate 클래스의 빈을 정의해보자. HibernateTemplate 클래스의 빈은 SessionFactory 인터페이스의 인스턴스가 필요하므로 위 예제에서 정의한 sessionFactory 빈을 연결하도록 다음처럼 선언해야 한다.

```
<bean id="hibernateTemplate"
      class="org.springframework.orm.hibernate.HibernateTemplate">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>

<bean id="movieDao"
      class="com.madhusudhan.jscore.data.MovieDAO">
  <property name="hibernateTemplate" ref="hibernateTemplate" />
</bean>
```

이제 정의 부분의 소스 코드 작성은 완료했으니 데이터를 가져오는 메서드 몇 개를 구현해보자. 다음 소스 코드에서 getMovie 메서드는 템플릿의 load 메서드를 사용한다.

```
public Movie getMovie(String id ){
  // Movie 객체를 검색해보자
  return (Movie)getHibernateTemplate().load(Movie.class, id);
}
```

이 소스 코드에서 확인할 수 있듯이 getMovie 메서드에서는 movie 객체 값을 가져오기 위해 SQL 구문을 사용하지 않았다. 데이터베이스를 다루는 것이 아니라 자바를 다룬다는 생각이 들 정도다! load 메서드에서는 데이터베이스에 접근한 다음 전달한 id를 사용해서 대응하는 행 데이터를 가져온다.

movie 객체를 업데이트하는 일도 역시 간단하다.

```
public void updateMovie(Movie m) {
  // Movie 객체를 업데이트하자
  getHibernateTemplate().update(m);
}
```

단 한 구문으로 모든 작업을 완료했다. 행 데이터를 삭제하는 일도 delete 메서드를 호출하면 되므로 매우 간단하다.

```
public void deleteMovie(Movie m){
  // Movie 객체를 삭제하자
  getHibernateTemplate().delete(m);
}
```

쿼리를 사용하는 일도 쉽다. 하이버네이트에서는 쿼리를 작성할 때 HQL[Hibernate Query Language]을 사용한다. 쿼리를 실행하려면 find 메서드를 사용한다.

예를 들어 id 값에 대응하는 Movie 객체를 반환하는 메서드를 다음 소스 코드로 확인해보자.

```
public Movie getMovie(String id) {
  String sql="from MOVIES as movies where movies.id=?";
  // Movie 객체를 찾자
  return (Movie)getHibernateTemplate().find(sql, id);
}
```

지금까지의 과정이 개괄적으로 살펴본 하이버네이트의 사용 방법이다. 이를 더 자세히 이해하려면 하이버네이트 관련 서적을 읽어보기 바란다.

네이티브 API 사용 방법

스프링을 하이버네이트와 함께 사용하는 방법 중 추천하는 것은 스프링의 템플릿 래퍼 클래스를 사용하는 것보다 하이버네이트의 네이티브 API를 사용하는 것이다. 『Just Spring Data Access』(『스프링 데이터 핵심 노트』로 출간할 예정)에서 자세히 설명했으니 나중에 이 책을 한번 읽어보기 바란다. 여기서는 간단히 살펴보고 지나가겠다.

org.hibernate.SessionFactory와 org.hibernate.Session 클래스는 하이버네이트 API의 중심을 이루고 있다. 스프링은 이 SessionFactory 인터페이스의 래퍼 클래스를 생성하기 위해 org.springframework.orm.hibernate3.LocalSessionFactoryBean 클래스를 제공한다. 이 래퍼 클래스는 XXXDAO 클래스 객체로 주입한다. 예상했듯이 이 클래스의 빈은 다른 하이버네이트 프로퍼티와 함께 데이터 소스와 연결해야 한다. XML 파일은 이전에 봤던 것과 거의 유사하다.

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="mySqlDataSource" />
  <property name="mappingResources">
    <list>
      <value>Trade.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.show_sql">false</prop>
      <prop key="hibernate.hbm2ddl.auto">true</prop>
    ....
    </props>
  </property>
</bean>
```

TradeDAO 클래스는 주입한 SessionFactory 인터페이스와 함께 여기에 선언했다.

```
public class TradeDAO {
  private SessionFactory sessionFactory = null;
  public SessionFactory getSessionFactory() {
```

```
    return sessionFactory;
  }
  public void setSessionFactory(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
  }
}
```

SessionFactory 인터페이스를 다룰 때의 장점은 getSessionFactory(). getCurrentSession 메서드를 호출해 session 객체를 가져올 수 있다는 것이다. Session은 데이터베이스 작업을 할 때 반드시 사용해야 하는 인터페이스다. 예를 들어 아래 t 객체를 영구 저장하려면 세션의 save 메서드만 호출하면 된다. 다음 소스 코드를 확인해보자.

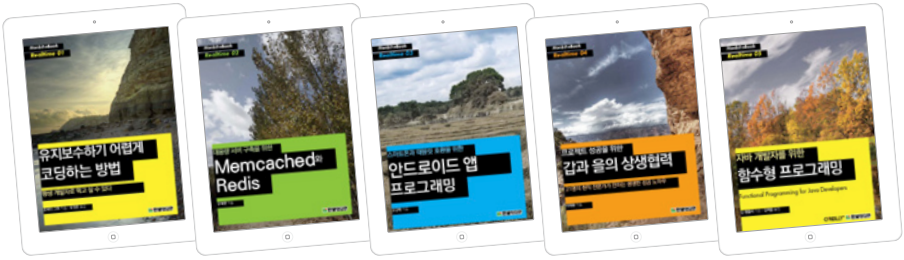
```
public void persist(Trade t) {
  session.beginTransaction();
  session.save(t);
  session.getTransaction().commit();
}
```

t 객체를 영구 저장하는 절차는 이게 전부다. 지저분한 SQL 구문이나 결과 집합 혹은 ResultSets 인터페이스는 더 이상 필요 없다. 각각의 데이터베이스 작업은 트랜잭션에서 수행해야 하므로 트랜잭션을 시작하고 이를 보내는 것으로 소스 코드를 작성해야 한다는 점은 유의하자.

이 외에도 Session 객체에서 실행할 수 있는 많은 작업이 있다. 이를 더 알고 싶다면 API 문서를 참조하도록 하자.

6.2 요약

6장에서는 스프링이 지원하는 JDBC와 하이버네이트를 소개했다. 예제에서 살펴봤듯이 스프링은 사용하기에 매우 간단하면서도 강력한 API를 제공함으로써 개발을 매우 편하게 해주었다. 즉, 많은 양의 반복 코드를 사용하기보다는 비즈니스 로직에만 집중할 수 있게 된 것이다.



01
유지보수하기 어렵게 코딩하는 방법

평생 개발자로 먹고 살 수 있다

–

로에디 그린 지음
우정은 옮김

이렇게 하면 개발자로 평생 먹고 살수 있다

02
대용량 서버 구축을 위한

Memcached와 Redis

–

강대명 지음

대용량 서버 구축을 위한 분산 캐시의 이해! 간단한 Short URL 실습으로 이해하는 분산 캐시 기술

03
스마트폰과 태블릿 호환을 위한

안드로이드 앱 프로그래밍

–

고강태 지음

스마트폰 앱을 태블릿 앱으로 쉽고, 빠르게 전환하고 싶으신가요? 다양한 기기와 호환하는 안드로이드 앱 개발의 모든 것!

04
프로젝트 성공을 위한 갑과 을의 상생협력

21명의 현직 전문가가 전하는 생생한 성공 노하우

–

이재용 지음

갑과 협력하기 어려우신가요? 을과 협력하기 어려우신가요? 21명의 현직 전문가가 알려주는 갑과 을의 상생협력 전략

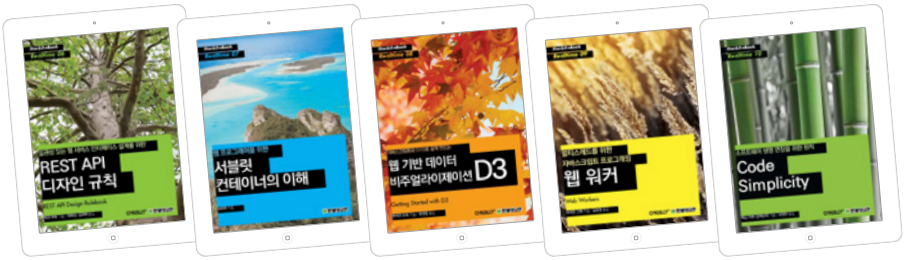
05
자바 개발자를 위한

함수형 프로그래밍

–

딘 왐플러 지음 / 입백준 옮김

자바 개발자을 위한 함수형 프로그래밍 기법 함수형 프로그래밍 기법을 익힌 프로그래머와 그렇지 않은 프로그래머가 작성하는 코드의 품질은 완전히 다르다



06
일관성 있는 웹 서비스 인터페이스 설계를 위한

REST API 디자인 규칙

–

마크 마세 지음
김관래, 권원상 옮김

REST API를 사용하기 어려우신가요? 사전처럼 찾아 쓰는 REST API 규칙과 팁

07
웹 프로그래머를 위한

서블릿 컨테이너의 이해

–

최희탁 지음

웹 프로그래밍에 깊이를 더 하자! 서블릿 컨테이너를 제대로 알면 웹 프로그래밍이 쉬워진다

08
자바스크립트와 SVG로 쉽게 만드는

웹 기반 데이터 비주얼라이제이션 D3

–

마이크 드워 지음
양성일 옮김

쉽게 배우는 웹 기반 데이터 비주얼라이제이션 D3

09
멀티스레드를 위한

자바스크립트 프로그래밍 웹 워커

–

아이두 그린 지음
김보경 옮김

웹 애플리케이션에서 멀티스레드를 구현하는 가장 쉬운 방법. 웹 애플리케이션에 날개를 달자!

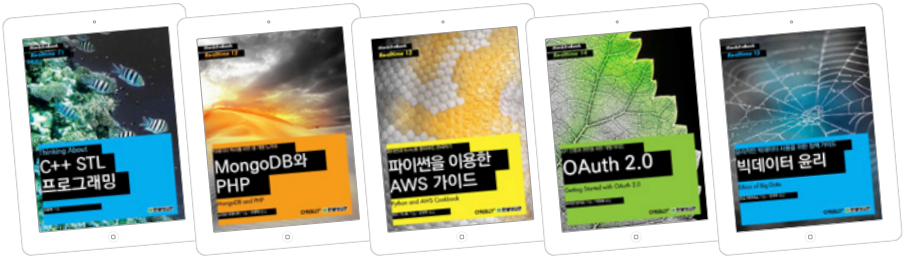
10
소프트웨어 생명 연장을 위한 원칙

Code Simplicity

–

맥스 카넷–알렉산더 지음
신정안 옮김

어떻게 해야 소프트웨어 설계를 잘할 수 있을까? 좋은 소프트웨어 설계는 무엇일까?



11

Thinking About
C++ STL 프로그래밍
–
최흥배 지음

C++ 프로그래밍에서는 STL은 필수다 STL을 아는 만큼 C++ 프로그래밍 스킬을 키울 수 있다

12

빅데이터 처리를 위한 웹 개발 노하우
MongoDB와 PHP
–
스티브 프랜시아 지음
최병현 옮김

MongoDB와 PHP의 만남, 빅데이터 처리 실전 노하우

13

파이썬과 Boto로 클라우드 관리하기
파이썬을 이용한 AWS 가이드
–
미치 가나트 지음
강권학 옮김

파이썬으로 AWS(아마존 웹 서비스)를 할 수 있다고요? 파이썬과 Boto로 AWS를 사용해보자

14

API 인증과 권한을 위한 개발가이드
OAuth 2.0
–
라이언 보이드 지음
이정림 옮김

웹 API 인증을 위한 만능 키, OAuth 2.0을 만나다

15

윤리적인 빅데이터 사용을 위한 정책 가이드
빅데이터 윤리
–
코드 데이비스 지음
강석주 옮김

빅데이터 시대! 윤리적인 책임은 없는가? 빅데이터 시대에서 지켜야 할 윤리 책임의 기준을 제시한다

16

윈도우 런타임을 이용한 실전 앱 개발
Windows 8 앱 개발 가이드
–
벤 듀이 지음 / 이원영 옮김

윈도우 런타임(Windows Runtime, WinRT)으로 시작하는 새로운 개발 세계

17

Xen으로 배우는 가상화 기술의 이해
CPU 가상화
–
박은병, 김태훈, 이상철, 문대혁 지음

반가상화와 전가상화 기술을 다루는 x86 아키텍처 기반의 CPU 가상화!

18

JSP 바이블
STEP 01
JSP 시작과 개발환경 구축
–
조효은 지음

JSP를 마스터하기 위한 길잡이 JSP를 위한 최고의 바이블 시리즈 첫 번째 책

19

빅데이터 시대의 개인정보보호와 사생활
프라이버시와 빅데이터
–
테렌스 크레이그, 메리 루틀로프 지음
이춘식, 김정환 옮김

빅데이터 시대! 개인정보와 프라이버시는 과연 보호받을 수 있을까?

20

프로그래머를 위한 가이드
드루팔
–
제니퍼 하지덴 지음
김지원 옮김

드루팔 개발을 위한 원칙과 팁 이렇게 하면 실수를 피할 수 있다