자바 개발자를 위한 97가지 제안

97 Things Every Java Programmer Should Know
by Kevlin Henney and Trisha Gee

Copyright © 2021 J-Pub Co., Ltd.

Authorized Korean translation of the English edition 97 Things Every Java Programmer Should Know,
ISBN 9781491952696 © 2020 O'Reilly Media Inc.

This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

이 책의 한국어판 저작권은 에이전시 원을 통해 저작권자와의 독점 계약으로 제이펍에 있습니다.
저작권법에 의해 한국 내에서 보호를 받는 저작물이므로 무단전재와 무단복제를 금합니다.

자바 개발자를 위한 97가지 제안

1쇄 발행 2020년 12월 24일

편집자 케블린 헤니, 트리샤 지
옮긴이 장현희
펴낸이 장성두
펴낸곳 주식회사 제이펍

출판신고 2009년 11월 10일 제406-2009-000087호
주소 경기도 파주시 회동길 159 3층 3-B호 / **전화** 070-8201-9010 / **팩스** 02-6280-0405
홈페이지 www.jpub.kr / **원고투고** submit@jpub.kr / **독자문의** help@jpub.kr / **교재문의** textbook@jpub.kr

편집팀 김정준, 이민숙, 최병찬, 이주원 / **소통 · 기획팀** 민지환, 송찬수, 강민철, 김수연 / **회계팀** 김유미
진행 장성두 / **교정 · 교열** 이미연 / **내지디자인 및 편집** 성은경 / **표지디자인** 이민숙
용지 신승지류유통 / **인쇄** 해외정판사 / **제본** 광우제책사

ISBN 979-11-90665-64-3 (93000)
값 22,000원

※ 이 책은 저작권법에 따라 보호를 받는 저작물이므로 무단 전재와 무단 복제를 금지하며,
이 책 내용의 전부 또는 일부를 이용하려면 반드시 저작권자와 제이펍의 서면동의를 받아야 합니다.
※ 잘못된 책은 구입하신 서점에서 바꾸어 드립니다.

제이펍은 독자 여러분의 아이디어와 원고 투고를 기다리고 있습니다. 책으로 펴내고자 하는 아이디어나 원고가 있는 분께서는 책의 간단한 개요와 차례, 구성과 저(역)자 약력 등을 메일(submit@jpub.kr)로 보내 주세요.

케블린 헤니, 트리샤 지 편집 장현희 옮김

※ 드리는 말씀

- 이 책에 기재된 내용을 기반으로 한 운용 결과에 대해 저자, 역자, 소프트웨어 개발자 및 제공자, 제이펍 출판사는 일체의 책임을 지지 않음으로 양해 바랍니다.
- 이 책에 등장하는 각 회사명, 제품명은 일반적으로 각 회사의 등록 상표 또는 상표입니다. 이 책에서는 ™, ©, ® 등의 기호를 생략하고 있습니다.
- 이 책에서 소개한 URL 등은 시간이 지나면 변경될 수 있습니다.
- 국내에 번역서가 출간된 원서는 번역서 제목과 국내 출판사 이름으로 변경하였습니다.
- 책의 내용과 관련된 문의 사항은 옮긴이나 출판사로 연락해 주시기 바랍니다.

– 옮긴이: aspnetmvp@gmail.com

– 출판사: help@jpub.kr

차 례

옮긴이 머리말 xi
서문 xiii
베타리더 후기 xvi

PROPOSAL 01 자바만으로도 충분하다 1
안데르스 노라스(*Anders Norås*)

PROPOSAL 02 확인 테스트 3
에밀리 배쉬(*Emily Bache*)

PROPOSAL 03 AsciiDoc으로 자바독 확장하기 5
제임스 엘리엇(*James Elliott*)

PROPOSAL 04 컨테이너를 제대로 이해하자 7
데이비드 델라바시(*David Delabassee*)

PROPOSAL 05 행위를 구현하는 것은 쉽지만 상태를 관리하는 것은 어렵다 9
에드슨 야나가(*Edson Yanaga*)

PROPOSAL 06 JMH로 조금 더 쉽게 벤치마킹해 보자 11
마이클 헝거(*Michael Hunger*)

PROPOSAL 07 아키텍처의 품질을 체계화하고 검증하는 방법의 장점 14
다니엘 브라이언트(*Daniel Bryant*)

PROPOSAL 08 문제와 업무를 더 작은 단위로 나누기 17
진 보야르스키(*Jeanne Boyarsky*)

PROPOSAL 09 다양성을 인정하는 팀 만들기 19
익셀 루이즈(*Ixchel Ruiz*)

PROPOSAL 10 빌드는 느려서도 안 되고 불안정해서도 안 된다 22
젠 스트레이터(*Jenn Strater*)

PROPOSAL 11 아니, 내 머신에서는 잘 실행됐다니까! 24
벤자민 무쉬코(*Benjamin Muschko*)

PROPOSAL 12 비대한 JAR은 이제 그만 27
다니엘 브라이언트(*Daniel Bryant*)

PROPOSAL 13 코드 복원전문가 29
에이브라함 마린-페레즈(*Abraham Marin-Perez*)

PROPOSAL 14 JVM의 동시성 31
마리오 푸스코(*Mario Fusco*)

PROPOSAL 15 CountDownLatch, 친구인가 적인가? 34
알렉세이 소신(*Alexey Soshin*)

PROPOSAL 16 선언적 표현식은 병렬성으로 가는 지름길이다 37
러셀 윈더(*Russel Winder*)

PROPOSAL 17 더 나은 소프트웨어를 더 빨리 전달하기 39
버크 허프나겔(*Burk Hufnagel*)

PROPOSAL 18 지금 몇 시예요? 41
크리스틴 고르만(*Christine Gorman*)

PROPOSAL 19 기본 도구의 사용에 충실하자 44
게일 올리스(*Gail Ollis*)

PROPOSAL 20 변수를 바꾸지 말자 46
스티브 프리먼(*Steve Freeman*)

PROPOSAL 21 SQL식 사고 도입하기 50
딘 웜플러(*Dean Wampler*)

PROPOSAL 22 자바 컴포넌트 간의 이벤트 52
마디 압델아지즈(*A.Mahdy AbdelAziz*)

PROPOSAL 23 피드백 루프 55
리즈 커프(*Liz Keogh*)

PROPOSAL 24 불꽃 그래프를 이용한 성능 확인 57
마이클 헝거(*Michael Hunger*)

PROPOSAL 25 지루하더라도 표준을 따르자 59
아담 베이언(*Adam Bien*)

PROPOSAL 26 자주 릴리스하면 위험을 줄일 수 있다 61
크리스 오델(*Chris O'Dell*)

PROPOSAL 27 퍼즐에서 제품까지 63
제시카 커(*Jessica Kerr*)

PROPOSAL 28 '풀스택 엔지니어'는 마음가짐이다 66
마체이 발코비악(*Maciej Walkowiak*)

PROPOSAL 29 가비지 컬렉션은 나의 친구 68
홀리 쿠민스(*Holly Cummins*)

PROPOSAL 30 이름 짓기를 잘 하자 70
피터 힐튼(*Peter Hilton*)

PROPOSAL 31 이봐 프레드, 해시맵 좀 전해 주겠는가? 72
커크 페퍼다인(*Kirk Pepperdine*)

PROPOSAL 32 널을 피하는 방법 74
카를로스 오브레건(*Carlos Obregón*)

PROPOSAL 33 JVM의 크래시를 유발하는 방법 77
토마스 론존(*Thomas Ronzon*)

PROPOSAL 34 지속적 전달로 반복가능성과 감사가능성 향상하기 79
빌리 코란도(*Billy Korando*)

PROPOSAL 35 자바는 자바만의 강점이 있다 81
제니퍼 레이프(*Jennifer Reif*)

PROPOSAL 36 인라인식 사고 83
패트리샤 애아스(*Patricia Aas*)

PROPOSAL 37 코틀린과의 상호운용 85
세바스티아노 포기(*Sebastiano Poggi*)

PROPOSAL 38 일은 끝났어요. 그런데... 87
진 보야르스키(*Jeanne Boyarsky*)

PROPOSAL 39 자바 자격증: 기술 업계의 터치스톤 89
말라 굽타(*Mala Gupta*)

PROPOSAL 40 자바는 90년대생 91
벤 에번스(*Ben Evans*)

PROPOSAL 41 JVM 성능 관점에서의 자바 프로그래밍 93
모니카 벡위드(*Monica Beckwith*)

PROPOSAL 42 자바는 재미있어야 한다 95
홀리 쿠민스(*Holly Cummins*)

PROPOSAL 43 자바의 불분명한 타입들 98
벤 에번스(*Ben Evans*)

PROPOSAL 44 JVM은 멀티패러다임 플랫폼이다 101
러셀 윈더(*Russel Winder*)

PROPOSAL 45 최신 동향을 파악하자 103
트리샤 지(*Trisha Gee*)

PROPOSAL 46 주석의 종류 105
니콜라이 팔로그(*Nicolai Parlog*)

PROPOSAL 47 은혜로운 flatMap 108
다니엘 이노호사(*Daniel Hinojosa*)

PROPOSAL 48 컬렉션을 제대로 이해하자 111
니킬 나니바디카라(*Nikhil Nanivadekar*)

PROPOSAL 49 코틀린은 정말 물건이다 113
마이크 던(*Mike Dunn*)

PROPOSAL 50 관용적인 자바 코드를 학습하고 머릿속에 캐시하자 117
진 보야르스키(*Jeanne Boyarsky*)

PROPOSAL 51 카타를 하기 위해 학습하고 카타를 이용해 학습하자 120
도널드 라브(*Donald Raab*)

PROPOSAL 52 레거시 코드를 사랑하는 방법 123
우베르토 바비니(*Uberto Barbini*)

PROPOSAL 53 새로운 자바 기능을 학습하자 126
게일 *C.* 앤더슨(*Gail C. Anderson*)

PROPOSAL 54 IDE를 활용해 인지 부하를 줄이는 방법 129
트리샤 지(*Trisha Gee*)

PROPOSAL 55 자바 API를 디자인하는 기술 131
마리오 푸스코(*Mario Fusco*)

PROPOSAL 56 간결하고 가독성이 좋은 코드 133
에밀리 장(*Emily Jiang*)

PROPOSAL 57 자바를 그루비스럽게 136
켄 쿠센(*Ken Kousen*)

PROPOSAL 58 생성자에서는 최소한의 작업만 140
스티브 프리먼(*Steve Freeman*)

PROPOSAL 59 Date라는 이름은 조금 더 명확해야 했다 143
케블린 헤니(*Kevlin Henney*)

PROPOSAL 60 업계의 발전에 기여하는 기술의 필요성 145
폴 *W.* 호머(*Paul W. Homer*)

PROPOSAL 61 바뀐 부분만 빌드하고 나머지는 재사용하기 147
젠 스트레이터(*Jenn Strater*)

PROPOSAL 62 오픈소스 프로젝트는 마법이 아니다 149
젠 스트레이터(*Jenn Strater*)

PROPOSAL 63 Optional은 규칙을 위반하는 모나드지만 좋은 타입이다 151
니콜라이 팔로그(*Nicolai Parlog*)

PROPOSAL 64 기본 접근 한정자를 가진 기능 단위 패키지 154
마르코 비렌(*Marco Beelen*)

PROPOSAL 65 프로덕션 환경은 지구상에서 가장 행복한 곳이다 157
조시 롱(*Josh Long*)

PROPOSAL 66 좋은 단위 테스트에 기반한 프로그래밍 ····· 160
케블린 헤니(*Kevlin Henny*)

PROPOSAL 67 OpenJDK 소스 코드를 매일 읽는 이유 ····· 163
하인츠 *M.* 카부츠(*Heinz M. Kabutz*)

PROPOSAL 68 내부를 제대로 들여다보기 ····· 165
라파엘 베네비지스(*Rafael Benevides*)

PROPOSAL 69 자바의 재탄생 ····· 168
샌더 맥(*Sander Mak*)

PROPOSAL 70 클로저에 의한 JVM의 재발견 ····· 170
제임스 엘리엇(*James Elliott*)

PROPOSAL 71 불리언 값은 열거자로 리팩토링하자 ····· 173
피터 힐튼(*Peter Hilton*)

PROPOSAL 72 속독을 위한 리팩토링 ····· 176
벤자민 무스칼라(*Benjamin Muskalla*)

PROPOSAL 73 단순한 값 객체 ····· 179
스티브 프리먼(*Steve Freeman*)

PROPOSAL 74 모듈 선언에 주의해야 하는 이유 ····· 182
니콜라이 팔로그(*Nicolai Parlog*)

PROPOSAL 75 의존성을 잘 관리하자 ····· 185
브라이언 베르메르(*Brian Vermeer*)

PROPOSAL 76 '관심사 분리'가 중요한 이유 ····· 187
데이브 팔리(*Dave Farley*)

PROPOSAL 77 기술 면접은 학습할 가치가 있는 기술이다 ····· 190
트리샤 지(*Trisha Gee*)

PROPOSAL 78 테스트 주도 개발 ····· 192
데이브 팔리(*Dave Farley*)

PROPOSAL 79 bin 디렉터리에는 좋은 도구가 너무나 많다 ····· 195
로드 힐튼(*Rod Hilton*)

PROPOSAL 80 자바 샌드박스를 벗어나자 ····· 198
이안 *F.* 다윈(*Ian F. Darwin*)

PROPOSAL 81 코루틴에 대한 고찰 ····· 201
던 그리피스(*Dawn Giffiths*), 데이비드 그리피스(*David Griffiths*)

PROPOSAL 82 스레드는 인프라스트럭처로 취급해야 한다 ····· 204
러셀 윈더(*Russel Winder*)

PROPOSAL 83 정말 좋은 개발자의 세 가지 특징 ····· 206
잔나 팟체이(*Jannah Patchay*)

PROPOSAL 84 마이크로서비스 아키텍처의 트레이드오프 208
케니 바타니(*Kenny Batani*)

PROPOSAL 85 예외를 확인하지 말자 210
케블린 헤니(*Kevlin Henny*)

PROPOSAL 86 컨테이너로 통합 테스트의 숨겨진 가능성을 끌어내자 213
케빈 위텍(*Kevin Wittek*)

PROPOSAL 87 퍼즈 테스트의 어마무시한 효과 215
냇 프래이치(*Nat Prayce*)

PROPOSAL 88 커버리지를 이용해 단위 테스트 개선하기 218
에밀리 배쉬(*Emily Bache*)

PROPOSAL 89 사용자 정의 아이덴티티 애노테이션을 자유롭게 사용하자 220
마크 리처드(*Mark Richards*)

PROPOSAL 90 테스트를 이용해 더 나은 소프트웨어를 더 빨리 개발하자 223
메릿 반 다이크(*Marit van Dijk*)

PROPOSAL 91 테스트 코드에 객체지향 원리 적용하기 225
앤지 존스(*Angie Jones*)

PROPOSAL 92 커뮤니티의 힘을 빌려 경력을 개발하자 228
샘 헵번(*Sam Hepburn*)

PROPOSAL 93 JCP 프로그램에 대한 이해와 참여 방법 230
헤더 반쿠라(*Heather VanCura*)

PROPOSAL 94 자격증에 가치를 두지 않는 이유 232
콜린 바이퍼스(*Colin Vipurs*)

PROPOSAL 95 주석은 한 문장으로 작성하라 234
피터 힐튼(*Peter Hilton*)

PROPOSAL 96 '읽기 좋은 코드'를 작성하자 237
데이브 팔리(*Dave Farley*)

PROPOSAL 97 젊은 객체, 늙은 객체, 그리고 가비지 240
마리아 아리아스 드 레이나(*Maria Arias de Reyna*)

기고자 소개 243
찾아보기 275

옮긴이 머리말

자바는 1995년 처음 등장한 이후 개선과 발전을 꾸준히 거듭해 오면서 현대 소프트웨어 개발 분야에 커다란 영향을 미치는 언어이자 플랫폼으로 성장했습니다. 기업은 물론 수많은 오픈소스 프로젝트가 자바로 개발되고 있으며, 여전히 많은 사람이 자바를 배워 IT 업계에 발을 들이고 있습니다.

개발자라면 충분히 이해하겠지만, 개발자로서의 경력을 처음 쌓기 시작하는 단계에서는 자바라는 언어 자체에 대한 이해, 더 나은 자바 코드를 작성하는 방법, 각종 프레임워크 등 자바를 언어로 바라보고 더 잘 이해하고 활용할 수 있는 방법에 집중하게 됩니다. 그러나 어느 정도 경험이 쌓이고 더 많은 업무와 책임을 갖게 되는 위치로 성장하게 되면 이제는 JVM의 동작 원리와 튜닝 방법, 더 높은 가독성과 유지보수성을 갖는 코드 작성, 성능과 확장성을 비롯해 애플리케이션의 아키텍처 설계 등으로 그 역량을 넓혀 나가게 되지요. 더불어 팀의 일원으로 많은 사람과 소통하고 협업하며 더 큰 규모의 프로젝트를 진행하고 완수하는 경험도 필요하게 됩니다.

이 책은 자바를 하나의 프로그래밍 언어로만 바라보지 않고 하나의 거대한 플랫폼과 생태계로 바라볼 수 있는 시야를 제공합니다. 풍부한 경험을 갖춘 73인의 기고자가 자신의 노하우를 통해 독자가 앞으로 자바 개발자로서 자신의 역량을 어떻게 발전시켜 나갈 것인지 다시 한번 생각하고 정리할 기회도 제공합니다. 이 책으로 고급 자바 프로그래밍 기술을 학습할 수 있는 것은 아니지만, 훌륭한 자바 개발자로 성장하기 위해 어떤 시각과 역량을 갖추어야 할지 되돌아보는 좋은 계기가 될 것입니다.

좋은 책을 번역할 수 있는 기회를 주신 장성두 대표님과 책의 편집과 출간에 힘을 보태 주신 제이펍 식구들에게 지면을 빌려 감사의 인사를 드립니다. 그리고 더 좋은 책을 만들 수 있도록 베타리딩에 참여해 원고를 검토하고 피드백을 남겨 주신 베타리더들에게도 고마움을 전합니다. 그리고 짧은 분량이지만 번역을 도와준 NHN 데이터플랫폼개발팀 오태겸 선임에게도 고마움을 표합니다. 평일에는 회사 일로, 주말에는 번역으로 늘 바쁜 남편과 아빠에게 투정보다는 응원을, 서운함보다는 든든한 지원과 애정을 표현해 준 아내 지영과 예린, 은혁에도 지면을 빌려 사랑하는 마음을 전합니다.

끝으로, 어려운 시기에도 개인의 역량 강화와 대한민국 소프트웨어 산업의 발전을 위해 늘 노력하는 독자 여러분을 응원합니다.

2020년 11월

옮긴이 **장현희** 드림

서 문

자신들의 지혜와 배려로 지금의 우리를 이끌어 준 모든 이를 기억하며...

"마음은 채워야 할 배가 아니라 태워야 할 장작이다."

— 플루타르코스(Plutarch)

모든 자바 프로그래머가 반드시 알아야 할 것은 무엇일까? 그 답은 누구에게 묻는지, 왜 묻는지, 언제 묻는지에 따라 다르다. 그리고 그 답도 저마다의 관점만큼이나 다양하다. 언어나 플랫폼, 생태계, 커뮤니티는 소프트웨어와 많은 사람의 삶에 영향을 미치며, 한 세기에서 다음 세기로, 하나의 코어에서 다중 코어로, 메가바이트에서 기가바이트로의 진화에도 영향을 미쳤던 이 질문의 답은 한 명의 저자가 하나의 책에 모두 담아 내기에는 너무 벅차다.

그래서 독자 여러분을 위해 자바 세계에 펼쳐진 수많은 관점과 표현을 이 책에 모아 봤다. 비록 모든 것을 다루고 있지는 않지만, 73명의 저자가 97가지 이야기를 담아 주었다. 이 책 《자바 개발자를 위한 97가지 제안》의 서문은 다음과 같이 시작하고자 한다.

> 알아야 할 것도, 해야 할 것도, 그 일을 수행하는 방법도 너무 많기에 어느 한 사람이나 하나의 논리도 '하나의 정답'을 주장할 수는 없다. 각기 다른 저자의 이야기는 반드시 일맥상통하지 않으며, 그럴 의도도 없다. 오히려 그 반대다. 저마다의 이야기는 그 독창성에 가치가 있다. 이 모음집의 가치는 각각의 저자가 자기 생각을 보완하고 확인하며 때로는 다른 저자의 생각에 반박하는 것에 있다. 누군가의 생각이 특별히 더 중요한 것도 아니다. 읽어 본 내

용에 반응하고, 이를 반영하며 서로 연관 짓고, 자기 생각과 지식, 경험에 기반해 어느 것에 더 무게를 둘 것인지는 오로지 독자 여러분에게 달렸다.

모든 자바 프로그래머가 알아야 할 것은 과연 무엇일까? 이 책에서 소개하는 97가지 해답은 언어, JVM, 테스팅 기술, JDK, 커뮤니티, 역사, 애자일식 생각, 구현 노하우, 전문성, 스타일, 본질, 프로그래밍 패러다임, 프로그래머를 사람의 관점에서 바라보는 시각, 소프트웨어 아키텍처, 코드 이상의 기술, 도구 사용 기술, GC 기법, 자바 외의 JVM 언어 등 다양한 분야로 확대되어 있다.

라이선스

최초의 '97가지(97 Things)' 도서의 정신을 이어받아 이번 책의 각 장은 제한 없는 오픈소스 모델을 따른다. 각 장은 크리에이티브 커먼즈 애트리뷰트 4.0 라이선스(https://oreil.ly/zPsKK)가 적용되어 있다. 또한, 이 중 상당수는 〈미디엄〉에 게시한 '97가지(https://medium.com/97-things)'에 처음 소개되었다.

이 모든 것이 독자 여러분의 생각과 코드의 발전을 위한 기틀이 되어 주길 바란다. 유튜브 http://youtube.com/oreillymedia에서도 볼 수 있다.

감사의 글

《자바 개발자를 위한 97가지 제안》은 많은 사람의 시간과 노력이 직간접적으로 투입된 프로젝트다. 모두 감사의 인사를 받아 마땅한 분들이다.

이 책에 수많은 시간과 노력을 들인 모든 이에게 지면을 빌려 감사의 인사를 전한다. 또한, 많은 피드백과 의견 및 제안을 제공해 준 브라이언 궤츠(Brian Goetz)에게도 감사를 전한다.

이 책을 집필하는 동안 여러 방면으로 이끌어주며, 기여자와 내용에 대한 지도편달을 아끼지 않은 잔 맥콰드(Zan McQuade)와 콜빈 콜린스(Corbin Collins) 이하 오라일리 출판사 관계자 여러분, 그리고 많은 힘을 보태 준 레이첼 루멜리오티스(Rachel Roumeliotis), 수잔 코난트(Susan Conant), 마이크 루키데즈(Mike Loukides)에게 고마움을 전한다.

케블린(Kevlin)은 쓸데없는 소리도 잘 들어준 그의 아내 케롤라인(Carolyn)과 늘 부모에게 힘이 되어준 두 아들 스테판(Stefan), 얀닉(Yannick)에게 감사 인사를 전하고 싶다고 한다.

그리고 트리샤(Trisha)는 뭔가를 충분히 하지 않는 것에 스트레스를 받는 것이 정작 뭔가를 하는 데 아무런 도움이 되지 않는다는 것을 알게 해준 남편 이스라(Isra)와 늘 조건 없는 사랑과 따듯한 포옹을 선사해 준 두 딸 에비(Evie)와 에이미(Amy)에게 감사를 전하고 싶다고 한다.

모쪼록 정보와 식견 그리고 영감을 얻으며 즐겁게 이 책을 읽어주길 바란다.

베타리더 후기

공민서(이글루시큐리티)

97가지 글 중 일부는 제목에서 자바를 제외해도 될 만큼 일반 개발자를 위한 범용적이고 알찬 노하우였습니다. 물론 자바에 특화된 내용도 있습니다. 설계, TDD 등 덜 실수하고 더 빠르게 성장하기 위한 선배들의 노하우를 모아 놓은 책이라 자바 개발자는 물론 다른 언어의 개발자들도 읽어보기를 추천합니다.

김진영(야놀자)

주제 하나하나가 너무 길지 않게 구성되어 출근 시간에 가볍게 읽기 좋은 책입니다. 자바를 처음 접해 보시는 분보다는 자바를 어느 정도 경험해 보았고 파편화된 지식을 다시 찾아보고 정리하고자 하는 분께 추천합니다. 전반적으로 꽤 재미있게 읽은 책이었습니다.

사지원(현대엠엔소프트)

자바 프로그래머만을 위한 책이 아닙니다. 이 책은 개발자로 살고 있는 사람이라면 누구나 공부했던 이야기를 하고 있습니다. 하지만 늘 부족한 시간과 빡빡한 일정에 많은 것을 놓치곤 합니다. 이 책은 우리가 '당연하게 알고 있던 것들'에 대해 다시 생각해 볼 기회를 제공해 줍니다. 특히 팀을 이뤄 업무를 진행하는 사람에게 강력히 추천합니다. 사실 에세이와 같은 책은 처음 리뷰해 보았습니다. 개발 서적이 아닌 책들에 대해 왠지 그릇된 편견이 있었는지도 모르겠습니다. 하지만 이 책을 보고 저의 잘못된 생각이 바로잡힌 기분이 듭니다. 팀을 이뤄 협업하는 자세를 다시 생각하고, 어떤 마음가짐으로 개발

업무를 대해야 하는지, 그리고 더 나은 개발자가 되기 위해선 어떤 방향으로 나아가야 하는지 정말 많은 것을 배웠습니다.

신진규(JEI)

다양한 자바 전문가의 이야기를 한 곳에서 들어볼 좋은 기회였습니다. 하지만 그들 모두는 같은 곳을 지향하진 않았습니다. 이 책 서문에서도 밝혔듯이 독자 자신의 상황에 따라 적절히 취사선택하면 좋겠습니다.

양성모(현대오토에버)

짧게 쓰겠습니다. 이전에 고민했던 것, 지금 고민하는 것, 앞으로 고민해야 할 것을 총망라하는 책이었습니다. 저 자신을 돌아보는 데 많은 도움이 되었습니다.

이석곤(엔컴)

선배 개발자가 중요하다고 생각하는 주제를 97가지로 나눠 정리한 책입니다. 글로벌 개발자 기준이라서 몇몇 주제는 한국 개발 문화와는 좀 맞지 않는 부분도 있었지만, 대체로 공감하는 부분이 많았습니다. 가장 기억이 남는 문구는 '좋은 개발자는 자격증이 필요 없다. 하지만 그렇지 않은 개발자라도 자격증은 쉽게 얻을 수 있다.' 이것은 한국이나 외국이나 같은 것 같습니다. 자바 교양도서로 편하게 읽어보면 좋겠습니다.

정현준(Agoda)

여러 개발 관련 주제는 물론 팀 문화, 데브옵스, CI/CD, 설계, 개발 도구 등 개발에 관련된 거의 모든 주제에 대한 노하우를 배울 수 있습니다. 당장 코드에 사용할 수 있을 만큼 실용적인 경우도 있고, 장기적으로 개발 문화를 수립하는 데 영감을 얻을 수도 있습니다. 처음에 훑어볼 때는 하나의 주제가 짧아서 크게 유용할까 의구심이 들었지만 베타리딩을 진행하면서 내용이 굉장히 마음에 들었습니다. 다만, 주제별로 묶었다면 좀 더 좋은 구성이 되었을 것으로 생각합니다.

차준성(서울아산병원)

장마다 멘토가 노하우를 알려주는 것 같은 친절한 책입니다. 시간이 날 때마다 교양서적처럼 읽기 좋습니다. 편집의 오류도 많이 없었고, 전반적으로 완성도가 높았던 책이었습니다.

제이펍은 책에 대한 애정과 기술에 대한 열정이 뜨거운 베타리더의 도움으로
출간되는 모든 IT 전문서에 사전 검증을 시행하고 있습니다.

01 PROPOSAL

자바만으로도 충분하다

안데르스 노라스(*Anders Norås*)

마이크로소프트(Microsoft)는 비주얼 스튜디오(Visual Studio)의 첫 메이저 개정본을 준비하는 동안 모트(Mort), 엘비스(Elvis), 아인슈타인(Einstein) 등 3명의 개발자 인격체를 세상에 소개했다.

모트는 매우 낙관적이어서 문제를 빠르게 해결하고 혼자서 질주하는 타입의 개발자였다. 엘비스는 실용적인 프로그래머로 자신의 업무에서 여전히 배우는 과정임에도 오래 지속될 솔루션을 구축하는 타입이었다. 마지막으로 아인슈타인은 약간 강박적인 프로그래머로, 코드를 작성하기 전에 가장 효율적인 솔루션을 디자인하고, 모든 것을 파헤치는 것에 집중하는 타입이었다.

프로그래밍 언어의 종교적 분열[1]에서 자바 진영에 속했던 우리는, 모트를 비웃으며 엘비스 같은 개발자가 '올바른 방법'으로 코드를 작성할 수 있게 하는 프레임워크를 만드는 아인슈타인 같은 개발자가 되길 원했다.

당시는 프레임워크의 시대가 도래하는 시점이었고 객체 관계형 매퍼(mapper)와 제어의 역전(Inversion of Control) 프레임워크를 능숙하게 다루지 못한다면 자바 프로그래머로 인정조차 받지 못하던 시기였다. 라이브러리는 규정된 아키텍처에 따라 프레임워크로 성장했다. 그리고 이 프레임워크가 기술 생태계를 이루어가면서 우리 중 상당수는 이렇게 성장할 수 있게 한 귀여운 언어 자바를 잊어가고 있다.

자바는 훌륭한 언어이며 자바 클래스 라이브러리는 범용으로 설계되었다. 파일을 다뤄야 한다면 `java.nio` 라이브러리를 쓰면 된다. 데이터베이스는 `java.sql` 라이브러리가 맡아 준다. 거의 모든 자바 배포판(distribution)은 완전한 기능의 HTTP 서버를 내장하고

1 역주: 자바/.Net을 의미하는 듯

있다. 물론 경우에 따라서는 자바라는 이름의 패키지 대신 `com.sun.net.httpserver` 패키지를 사용해야 하는 경우도 있지만 말이다.

애플리케이션이 하나의 함수를 배포 단위로 사용하는 서버리스(serverless) 아키텍처로 이동하면서 애플리케이션 프레임워크의 장점들은 희석되고 있다. 그 이유는 기술 및 아키텍처 관점의 문제들을 처리하는 시간은 줄어들고 프로그램의 비즈니스적 기능에 프로그래밍 노력을 더 많이 들일 수 있게 되었기 때문이다.

브루스 조이스(Bruce Joyce)는 이렇게 표현했다.

> 우리는 때때로 바퀴를 재발명해야 한다. 많은 수의 바퀴가 필요하기보다는, 많은 수의 발명가가 필요하기 때문이다.

많은 개발자가 재사용성을 극대화하기 위해 범용 비즈니스 로직 프레임워크의 개발에 착수했다. 하지만 범용 비즈니스 문제란 사실상 존재하지 않았으므로 대부분 실패로 돌아갔다. 뭔가 특별한 것을 특별한 방법으로 실행하는 것이야말로 어떤 한 비즈니스를 다른 비즈니스와 차별화하는 것 아닌가. 그러므로 프로젝트마다 새로운 비즈니스 로직을 작성할 일이 있는 것이다. 범용적이며 재사용할 수 있다는 관점에서 보면 일종의 룰 엔진(rule engine) 같은 것을 도입하고 싶어지기도 한다. 하지만 결국 룰 엔진을 설정하는 것도 프로그래밍이며, 이 경우 대부분은 자바보다 못한 언어를 사용하게 된다. 왜 그냥 자바로 코드를 쓰지 않을까? 자바로 작성한 코드는 읽기 쉽고, 심지어 자바 프로그래머가 아니어도 쉽게 유지보수할 수 있는 코드를 산출할 수 있다.

종종 자바의 클래스 라이브러리가 조금 제한적이라는 것을 느끼고 날짜나 네트워킹 같은 것을 조금 더 편하게 다룰 수 있는 다른 뭔가를 갈구할 때도 있을 것이다. 그래도 괜찮다. 필요한 라이브러리를 사용하면 된다. 다른 점이라면 독자 여러분이 매일 사용하는 기술 스택의 일부이기 때문에 사용하는 것이 아니라 구체적인 필요 때문에 라이브러리를 사용한다는 점이다.

다음에 문득 작은 프로그램에 대한 아이디어가 떠오른다면, JHipster(https://www.jhipster.tech)가 만들어주는 코드를 찾지 말고 잠들어 있던 자바 클래스 라이브러리에 대한 지식을 깨워 내길 바란다. **유행을 좇지 말자.** 이제는 단순함의 시대다. 모트도 이런 삶을 좋아했을 것이라고 확신한다.

PROPOSAL

02 확인 테스트

에밀리 배쉬(*Emily Bache*)

다음 코드처럼 빈 값이나 의미 없는 값을 확인하는 테스트 코드를 작성해 본 적이 있는가?

```
assertEquals("", functionCall())
```

보통 `functionCall` 함수가 문자열을 리턴하는데 이 문자열이 정확히 어떤 값이어야 되는지 모르지만 리턴값을 보면 맞는지 아닌지 알 수 있는 경우에 이런 코드를 작성한다. 물론 `functionCall` 함수는 빈 문자열이 아닌 어떤 문자열을 리턴할 것이므로 처음 테스트를 실행하면 실패하게 된다(어쩌면 올바른 리턴값을 볼 때까지 몇 번 더 실행해 볼지도 모르겠다). 그런 다음, 리턴값을 복사해서 `assertEquals` 함수 파라미터로 복사해 넣는다. 이제 테스트를 다시 실행해 보면 통과할 것이다. 이걸로 끝! 필자는 이 방법을 **확인 테스트(approval testing)**라고 부른다.

여기서 가장 중요한 단계는 일단 출력이 올바른지 확인한 후 이를 기댓값(expected value)으로 사용하는 부분이다. 코드 작성자가 결과를 '확인'했으므로 그 값을 테스트에 사용해도 무방하다. 대부분 독자는 이 방법을 깊이 생각할 겨를없이 사용하고 있었을 것이다. 어쩌면 이 방법을 **스냅숏 테스트(snapshot testing)**나 **골든 마스터 테스트(golden master testing)** 같은 이름으로 알고 있을지도 모르겠다. 필자의 경험상, 이 방법을 지원하도록 디자인된 테스트 프레임워크를 사용하고 있다면 많은 부분을 분명하게 이해할 수 있고 이 방법을 활용하면 훨씬 더 쉽게 테스트할 수 있다.

JUnit 같은 단위 테스트 프레임워크를 사용하면 함수의 리턴 문자열이 변경될 때 테스트를 수정하기가 어려울 수 있다. 결국 소스 코드 여기저기서 변경된 기댓값을 복사해 붙여 넣게 된다. 하지만 확인 테스트 도구를 사용하면 확인한 문자열이 파일에 대신 저장된다.

이 방법은 새로운 가능성을 열어준다. 예를 들면 비교 도구(diff tool)를 열어 변경 사항을 확인한 후 하나씩 머지(merge)하면 된다. JSON 문자열 같은 것을 다룰 때는 문법 강조(syntax highlighting) 지원도 받을 수 있다. 게다가 각기 다른 클래스에 대한 여러 테스트를 찾아 바꾸기 기능으로 한 번에 수정할 수도 있다.

그러면 확인 테스트는 어떤 경우에 활용할 수 있을까? 다음 예를 살펴보자.

변경해야 할 단위 테스트가 없는 코드

코드가 프로덕션 환경에서 실행 중이면 기본적으로 코드의 모든 동작은 올바르며 확인된 것으로 간주한다. 이런 코드에 대한 테스트를 만들 때 어려운 점은 테스트를 작성하는 작업이 확인해야 할 데이터를 리턴하는 로직의 경계를 찾아 이를 분리하는 문제로 탈바꿈한다는 점이다.

JSON이나 XML을 리턴하는 REST API와 함수

결과 문자열이 길다면 소스 코드 외부에 저장하는 편이 좋다. JSON과 XML은 모두 공백을 이용해 형식화되었으므로 기댓값과 비교하기도 쉽다. 만일 JSON이나 XML에 변화가 큰 값–예를 들면 날짜와 시간–이 담겨 있다면 나머지 값들을 확인하기 전에 이 값들을 별도로 검사해야 한다.

복합 객체를 리턴하는 비즈니스 로직

복합 객체(complex object)를 전달받아 이를 문자열로 출력하는 `Printer` 클래스를 생각해 보자. 또는 `Receipt`나 `Prescription` 또는 `Order` 클래스도 좋은 예다. 이 데이터는 사람이 읽을 수 있는 여러 줄의 문자열로 표현할 수 있다. `Printer` 클래스는 객체 그래프를 살펴보고 관련된 상세 내용만 출력하는 식으로 요약 정보만 출력하도록 구현할 수도 있다. 그러면 테스트 코드는 여러 비즈니스 규칙을 테스트하며 `Printer` 클래스를 이용해 확인할 수 있는 문자열을 만들어 낼 수 있다. 그렇게 하면 코딩 경험이 없는 제품 소유자(product owner)나 비즈니스 분석가(business analyst)라도 테스트 결과를 읽고 결과의 올바름 여부를 판단할 수 있다.

이미 한 줄 이상의 문자열을 확인하는 테스트를 작성했다면 확인 테스트를 더 알아보고 이를 지원하는 도구를 사용해 보길 권한다.

03

PROPOSAL

AsciiDoc으로 자바독 확장하기

제임스 엘리엇(*James Elliott*)

자바 개발자라면 자바독(Javadoc)을 이미 알고 있을 것이다. 자바는 컴파일러와 표준 도구에 문서 생성기(documentation generator)를 직접 통합한 최초의 주류 언어였기에 자바를 오래 사용한 개발자라면 자바독이 얼마나 잘 문서로 변환되는지도 기억할 것이다. 자바독은 (간혹 매끄럽지 않거나 썩 훌륭하진 않지만) 풍부한 API 문서를 생성한다는 점에서 큰 이점이 있으며 이런 트렌드는 다른 많은 언어에도 퍼져 나갔다. 자바독은 제임스 고슬링(James Gosling)의 토론(https://oreil.ly/Y_7rk) 내용처럼 '자바독보다는 좋은 기술 문서 작성자가 훨씬 더 나은 결과물을 만들 수 있었기'에 처음에는 매우 보수적이었지만, 기술 문서 작성자가 문서화하기에는 API의 수가 너무 많아서 보편적으로 사용할 수 있는 도구의 가치가 높아졌다.

그런데 간혹 자바독이 제공하는 API 문서, 즉 패키지와 프로젝트 개요 페이지 이외의 것이 필요하다. 최종 사용자를 위한 가이드나 따라 하기, 아키텍처와 이론에 대한 상세한 배경지식, 여러 컴포넌트가 어떻게 함께 어우러져 기능을 완성하는지에 대한 설명 등, 이 중 어느 것도 자바독만으로 충분히 문서화할 수 없다.

그러면 어떤 도구로 이런 수요를 맞춰 줄 수 있을까? 그 답은 시간이 흐르면서 계속 바뀌어 왔다. 80년대에는 GUI 기반 크로스 플랫폼 기술 문서 도구인 프레임메이커(FrameMaker)가 대세였다. 자바독 역시 매력적인 API 문서를 생성하기 위해 프레임메이커의 MID 독릿(Doclet)을 포함하곤 했지만 이제는 기능이 뒤떨어지는 윈도우(Windows) 버전만 남아 있다. 독북(DocBook) XML은 공개 표준에 기반하며 프레임메이커와 유사한 구조 및 문서 연결 기능을 지원하는 크로스 플랫폼 도구를 제공하지만, XML 형식을 직접 다루는 것은 실용적이지 않았다. 게다가 편집 도구를 계속 실행해야 하는 것도 문제

였지만, 그나마 나은 도구조차도 조악하고 집필에 방해가 되는 경우가 많았다.

하지만 마침내 훨씬 나은 대안을 찾았다. 바로 아스키독(AsciiDoc, https://oreil.ly/NYrJI)이다. 아스키독은 독북처럼 작성하기 쉽고 읽기 쉬운 텍스트 형식을 채택해서 간단한 작업은 쉽게 처리할 수 있으면서도 복잡한 작업 또한 가능하다. 대부분 아스키독 구조는 온라인 토론 포럼에서 많이 사용하는 마크다운(MarkDown) 같은 경량(lightweight) 마크업 형식처럼 즉각 읽을 수 있는 구조다. 문서를 더 예쁘게 꾸미고 싶으면 MathML이나 LaTeX 형식을 이용해 복잡한 수식을 포함하거나 텍스트 문단에 연결되며 줄번호까지 첨부된 소스 코드, 여러 종류의 경고 블록(block) 등도 추가할 수 있다.

아스키독은 2002년 파이썬으로 처음 개발되었다. 현재 공식 구현체는 2013년에 출시된 아스키닥터(Asciidoctor, https://oreil.ly/aRRvG)다. 이 도구의 루비(Ruby) 코드는 아스키닥터J(AsciidoctorJ, https://oreil.ly/UT8EP)를 통해 JVM에서도 실행되거나(메이븐(Maven) 또는 그레이들(Gradle) 플러그인이 필요하다) 자바스크립트로 변환(https://oreil.ly/E_6qn)할 수 있으며, 지속적 통합 환경에서 자연스럽게 동작한다. 안토라(Antora, https://antora.org) 같은 도구를 이용하면 (심지어 여러 리포지토리로부터) 관련된 문서를 한데 모은 완전한 사이트도 놀랍도록 쉽게 만들 수 있다. 커뮤니티(https://oreil.ly/PtWwa)도 친절하고 협조적이며 지난 몇 년간 이 도구가 성장하고 발전하는 과정을 지켜 본 경험도 꽤 인상적이었다. 또한, 관심 있는 독자를 위해 예전 아스키독 명세의 표준화 작업이 진행 중이라는 점도 언급하고 싶다(https://oreil.ly/BaXa8).

필자는 다른 사람과 공유하는 프로젝트에 풍부하고 매력적인 문서(https://oreil.ly/H_rSW)를 지원하는 것을 좋아한다. 아스키독 덕분에 훨씬 쉽게 문서화할 수 있었으며 작업 주기를 빠르게 가져갈 수 있었다. 덕분에 문서를 더 완벽하고 보기 좋게 꾸미는 작업이 훨씬 재미있어졌다(https://oreil.ly/7sbtj). 바라건대 독자 여러분도 필자와 같은 재미를 찾길 바란다. 또한 모든 문서를 아스키독으로 작성하기로 했다면 아스키독을 이용해 자바독을 작성할 수 있게 도와주는 독릿(https://oreil.ly/9KgQq)도 살펴보길 바란다.

24

PROPOSAL

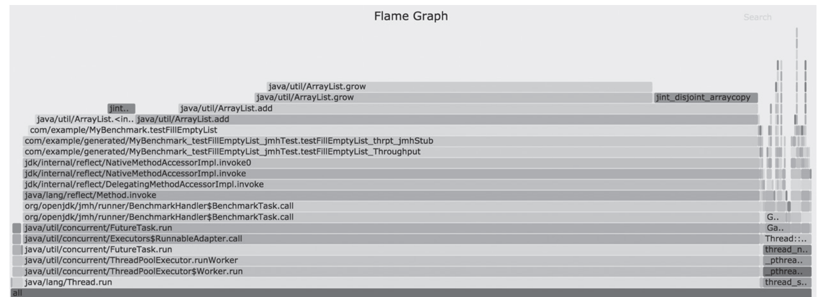
불꽃 그래프를 이용한 성능 확인

마이클 헝거(*Michael Hunger*)

보편적인 자바 프로파일러(profiler)는 바이트 코드 계측(byte code instrumentation)이나 샘플링(짧은 주기로 스택 트레이스(stack trace)를 수집하는 방법) 기법을 이용해 실행 시간이 긴 구간을 찾는다. 두 방법 모두 각자의 편법이 동원된다. 프로파일러의 출력을 이해하는 것은 그 자체로 예술이며 경험이 많이 필요하다.

다행히 넷플릭스(Netflix)의 성능 엔지니어인 브렌든 그렉(Brendan Gregg, https://oreil.ly/dhd5O)이 거의 모든 시스템으로부터 스택 트레이스를 수집해 기발한 형태의 다이어그램으로 보여주는 **불꽃 그래프(flame graphs, https://oreil.ly/2kCDd)**를 만들어냈다.

불꽃 그래프는 트레이스를 각 스택 수준으로 정렬해 집계한 것으로 각 스택 수준별 카운트는 코드의 각 부분의 총 실행 시간의 백분율을 의미한다. 이 백분율을 너비로 하는 블록(사각형)으로 렌더링하고 각 블록을 쌓아보면 매우 유용한 결과를 볼 수 있다.



'불꽃'은 아래에서 위로 향하며 프로그램이나 스레드(`main` 또는 이벤트 루프)의 진입점부터 코드가 실행되는 과정을 거쳐 불꽃의 끝에서는 코드 실행이 완료되는 지점을 표현한다.

왼쪽부터 오른쪽으로의 순서는 그다지 중요하지 않다. 대부분 알파벳순으로 정렬할 뿐이다. 색상도 크게 중요하지 않다. 중요한 것은 각 블록의 너비와 스택의 깊이(depth)다.

이 그래프를 보면 프로그램에서 예상보다 많은 시간을 소비하는 부분을 즉각 알아볼 수 있다. 그래프의 높이가 높을수록 더 많은 시간을 허비한 것이다. 특히 꼭대기 부분에서 매우 넓은 블록을 본다는 것은 해당 부분에서 병목이 발생하고 있다는 뜻이다. 이슈를 해결하고 다시 측정해 보자. 만일 전체적인 성능 이슈가 계속된다면 새로운 조짐이 드러나는지 다이어그램을 다시 확인하자.

보편적인 프로파일러의 단점을 극복하기 위해 최근의 도구들은 안전점(safepoint) 외부에서 스택 트레이스를 수집할 수 있는 JVM의 내장 기능(`AsyncGetCallTrace`)을 사용한다. 게다가 JVM 작업의 측정 결과를 네이티브 코드 및 운영체제로의 시스템 콜과 결합해서 보여주므로 네트워크에서 소비된 시간, 입출력, 가비지 컬렉션(garbage collection) 등에 사용된 시간도 불꽃 그래프에 함께 표시된다.

어니스트 프로파일러(Honest Profiler), perf-map-agent, async-profiler 같은 도구는 물론 심지어 인텔리제이(IntelliJ) IDEA 쉽게 정보를 수집해 불꽃 그래프를 생성한다.

대부분 해당 도구를 다운로드하고 자바 프로세스의 프로세스 ID(PID)를 지정한 후 해당 도구를 일정 시간 실행하면 대화형 SVG 형식으로 결과를 얻을 수 있다.

```
# https://github.com/jvm-profiling-tools/async-profiler에서
# async profilder를 다운로드해서 압축 해제한다.
./profiler.sh -d <duration> -f flamegraph.svg -s -o svg <pid> && \
open flamegraph.svg -a "Google Chrome"
```

이 도구가 생성한 SVG는 색상이 적용되었을 뿐 아니라 대화형으로 확인할 수 있다. 예를 들어 어떤 섹션을 확대하거나 심볼을 검색하는 등의 기능이 제공된다.

불꽃 그래프는 프로그램의 전반적인 성능 특성을 쉽게 확인할 수 있는 매우 강력한 도구다. 문제가 있는 부분을 즉시 확인하고 집중할 뿐 아니라 더 많은 정보를 제시하기 위해 JVM 이외의 시스템 특성값도 포함한다.

25

PROPOSAL

지루하더라도 표준을 따르자

아담 베이언(*Adam Bien*)

자바 초창기에는 시장에 호환성이 없는 애플리케이션 서버가 많았으며 서버 벤더는 완전히 다른 패러다임을 따르고 있었다. 일부 서버는 심지어 C++ 같은 네이티브 언어로 구현된 부분도 있었다. 여러 서버를 모두 이해하는 것은 어려웠고 애플리케이션을 한 서버에서 다른 서버로 이전하는 것은 불가능에 가까웠다.

JDBC(JDK 1.1에서 도입), JNDI(JDK 1.3에서 도입), JMS, JPA, 서블릿 같은 API가 등장하면서 이미 구현된 제품의 추상화, 간결화, 통합이 이루어졌다. EJB와 CDI 덕분에 배포와 프로그래밍 모델이 벤더와 무관해졌다. 처음 J2EE로 등장해서 Java EE로, 지금은 자카르타(Jakarta) EE가 된 기술과 마이크로프로파일(Microprofile)은 애플리케이션 서버가 구현해야 할 최소한의 API 집합을 정의했다. J2EE의 장점 덕분에 개발자는 J2EE API만 알면 애플리케이션을 개발하고 배포할 수 있었다.

서버가 개선되어도 J2EE와 Java EE API는 여전히 호환되었다. 새로운 애플리케이션 서버가 릴리스된다고 해서 애플리케이션을 마이그레이션할 필요가 없었다. 심지어 Java EE를 더 높은 버전으로 업그레이드하기도 쉬웠다. 애플리케이션을 다시 컴파일할 필요도 없이 테스트만 다시 하면 그만이었다. 애플리케이션의 리팩토링도 새로운 API의 장점을 적용하고 싶은 경우에만 하면 됐다. J2EE를 도입함으로써 개발자들은 세부 명세를 깊이 파고들지 않아도 여러 애플리케이션 서버들을 활용할 수 있었다.

현재 웹/자바스크립트 생태계도 유사한 상황에 처해 있다. 제이쿼리(jQuery), 백본(Backbone.js), 앵귤러JS 1, 앵귤러2+ (앵귤러JS 1과는 완전히 다르다), 리액트JS, 폴리머(Polymer), 뷰(Vue.js), 엠버(Ember.js) 등의 프레임워크는 완전히 다른 규칙과 패러다임을

따른다. 그래서 한 번에 여러 프레임워크를 학습하기가 어렵다. 많은 프레임워크의 본질적인 목적은 서로 다른 브라우저 사이의 호환성 이슈를 해결하기 위한 것이었다. 하지만 브라우저가 놀라운 수준으로 호환성을 높이면서 프레임워크는 데이터 바인딩, 단방향 데이터 흐름(unidirectional data flow)은 물론 의존성 주입(dependency injection) 같은 엔터프라이즈 자바 기능까지 제공하기 시작했다.

동시에 브라우저는 호환성을 갖췄을 뿐 아니라 예전에는 서드파티 프레임워크에서만 지원하던 기능까지 지원하기 시작했다. `querySelector` 기능은 이제 모든 브라우저에서 사용할 수 있으며 제이쿼리의 DOM 접근 기능과 호환되는 기능을 제공한다. 사용자 정의 요소(custom element)를 이용한 웹 컴포넌트, 쉐도우 DOM, 템플릿 등의 기능 덕분에 개발자는 새로운 요소를 정의해 UI와 동작을 구현할 수 있음은 물론 이를 애플리케이션 전체 구조로까지 확대할 수 있다. ECMA 스크립트 6부터 자바스크립트는 자바와 더 유사해졌으며 ES6 모듈은 번들링(bundling)을 선택적으로 수행할 수 있게 바뀌었다. MDN(Mozilla Developer Network)은 구글, 마이크로소프트, 모질라, W3C, 삼성 등 웹 표준에 관여하는 기업이 모두 힘을 합쳐 웹 표준과 관련한 정보를 제공하는 서비스가 되었다.

이제는 프레임워크 없이도 프론트엔드를 구현할 수 있다. 브라우저의 하위 호환성은 점점 더 훌륭해지고 있다. 어쨌든 모든 프레임워크는 브라우저 API를 사용하므로 표준을 학습하면 프레임워크를 더 잘 이해할 수 있다. 새 버전 브라우저에 브레이킹 체인지(breaking change)가 등장하지 않는 한 프레임워크 없이 웹 표준에만 의지해도 애플리케이션을 구현할 수 있다.

표준에 집중하면 시간이 지나면서 더 많은 지식을 확보할 수 있다. 매우 효율적인 학습 방법이다. 대중적인 프레임워크를 활용해 보는 것은 재미있겠지만 그렇게 얻은 지식은 다음에 다른 '인기 기술'이 등장하면 소용없는 지식이 되어버린다.

57

PROPOSAL

자바를 그루비스럽게

켄 쿠센(*Ken Kousen*)

화면에 나타난 것은 첫 줄부터 무슨 사이버펑크 소설을 펼친 것 같은 느낌이었다. 가만히 바라보다가 오늘밤까지 절대 끝내지 못할 거라는 걱정이 들었다. 갑자기 좁디좁은 내 방의 문을 누군가 두드린다. 문을 열어보니 보스가 서 있었다.

"잘 돼 가나요?" 그녀가 물었다.

"자바는 너무 장황해요." 나는 한숨을 쉬며 말을 이었다. "그냥 서비스에서 데이터를 다운로드해서 데이터베이스에 저장하고 싶을 뿐이에요. 그런데 빌더며 팩토리며 라이브러리 코드에 try/catch 블록에... 난리도 아니예요."

"그냥 그루비를 추가해요."

"예? 그게 무슨 도움이 된다고요?" 그러자 보스가 직접 컴퓨터 앞에 앉으며 물었다.

"내가 해도 되죠?"

"그럼요."

"간단한 데모를 하나 보여줄게요."라며 명령 프롬프트를 열더니 groovyConsole을 입력한다. 그러자 화면에 간단한 GUI가 나타났다. "지금 우주에 우주비행사가 몇 명이나 나가 있는지 알고 싶다고 생각해 봅시다. 오픈 노티파이(Open Notify, https://oreil.ly/oysGk) 서비스를 호출하면 그 답을 알 수 있어요."

그러더니 그루비 콘솔에 다음 명령을 입력했다.

```
def jsonTxt = 'http://api.open-notify.org/astros.json'.toURL().text
```

그러자 우주비행사의 수, 상태 메시지, 우주비행선과 각 우주비행사의 관계를 표현하는 중첩된 객체를 담은 JSON 응답이 화면에 나타났다.

"그루비에는 String을 java.net.URL 타입으로 바꾸는 toURL 함수가 추가되었어요. 그리고 URL에 추가한 getText 메서드는 데이터를 조회해서 텍스트로 리턴하죠."

"끝내주네요."라며 맞장구쳤다. "이제 이걸 자바 클래스로 매핑해야 하니까 Gson이나 Jackson 같은 라이브러리를 쓰는 게…"

"뭐라는 거예요. 그냥 우주에 몇 명이나 있는지 알고 싶은 거니까 이럴 때는 JsonSlurper를 쓰면 돼요."

"아, 뭐라고요?"

그러자 보스가 다음 명령을 입력했다.

```
def number = new JsonSlurper().parseText(jsonTxt).number
```

"parseText 메서드는 Object를 리턴한다고요." 보스가 설명을 이어갔다. "근데 우린 타입은 관심이 없으니까 그냥 필요한 데이터를 바로 읽으면 돼요."

결과를 보니 현재 우주 공간에는 6명의 우주비행사가 모두 국제 우주 공항에 머물고 있음을 알 수 있었다.

"알았어요, 근데 만약 이 응답을 클래스로 파싱하고 싶으면 어떻게 해요? 그루비용 Gson 라이브러리가 있나요?"

보스는 고개를 가로저었다. "그런 건 필요 없어요. 그루비도 결국은 바이트코드라고요. 그냥 하던 대로 Gson 클래스 인스턴스를 만들어서 메서드를 호출하면 되죠."

```
@Canonical
class Assignment { String name; String craft }
@Canonical
class Response { String message; int number; Assignment[] people }
new Gson().fromJson(jsonTxt, Response).people.each { println it }
```

“Canonical 애노테이션은 클래스에 toString, equals, hashCode, 기본 생성자, 매개변수 생성자, 튜플 생성자 등을 클래스에 알아서 구현해 주죠.”

“엄청나네요! 그럼 이제 우주비행사를 데이터베이스에 어떻게 저장하죠?”

“껌이죠. H2에 저장해 볼까요?”

```
Sql sql = Sql.newInstance(url: 'jdbc:h2:~/astro',
                          driver: 'org.h2.Driver')
sql.execute '''
    create table if not exists ASTRONAUTS(
        id int auto_increment primary key,
        name varchar(50),
        craft varchar(50)
    )
'''
response.people.each {
    sql.execute "insert into ASTRONAUTS(name, craft)" + ($it.name, $it.craft)" +
                "values ($it.name, $it.craft)"
}
sql.close()
```

“그루비의 Sql 클래스와 여러 줄 문자열로 테이블을 생성하고 문자열 연결을 이용해서 값을 인서트하면 금방이죠.”

```
sql.eachRow('select * from ASTRONAUTS') {
    row -> println "${row.name.padRight(20)} aboard ${row.craft}"
}
```

“다 됐어요.” 그녀가 말했다. “게다가 출력까지 예쁘게 잘됐네요.”

나는 결과물을 물끄러미 바라봤다. 그리고는 “이걸 자바로 하려면 대체 몇 줄이나 코드를 짜야 하는지 알아요?”라고 물었다.

그러자 그녀가 씩 웃으며 말했다. “장난 없겠죠. 근데 그루비가 던지는 예외는 전부 확인되지 않은(unchecked) 거라 try/catch 블록도 필요 없어요. 그리고 newInstance 대신 withInstance 메서드를 쓰면 DB 연결도 자동으로 닫혀요. 좋죠?”

나는 고개를 끄덕이며 동의했다.

“이 코드를 클래스로 나눠 감싸면 자바에서도 호출할 수 있어요.”

그렇게 그녀는 자리를 떴고 나는 자바를 더 그루비스럽게 만들 수 있다는 기대감에 가득 찼다.

58 생성자에서는 최소한의 작업만

PROPOSAL

스티브 프리먼(*Steve Freeman*)

내가 자주 보는 패턴 중 하나는 생성자(constructor)에서 대부분 작업을 처리하는 패턴이다. 일련의 인수(argument)를 전달받아 각 필드의 값으로 바꾸는 것이다. 대략적인 형태는 다음과 같다.

```
public class Thing {
    private final Fixed fixed;
    private Details details;
    private NotFixed notFixed;
    // 그 외의 필드

    public Thing(Fixed fixed,
                 Dependencies dependencies,
                 OtherStuff otherStuff) {
        this.fixed = fixed;
        setup(dependencies, otherStuff);
    }
}
```

예상컨대 `setup` 메서드는 `dependencies`와 `otherStuff` 매개변수로 나머지 필드를 초기화할 것이다. 하지만 대체 생성자 시그너처가 새로운 인스턴스를 생성하는 데 어떤 이점을 가져다주는지 모르겠다. 게다가 객체의 수명주기 동안 어떤 필드가 변경되는지도 명확하지 않다. 필드를 생성자에서 초기화하지 않는 한 `final`로 선언할 수 없기 때문이다. 마지막으로 이 클래스는 인스턴스를 생성하기 위해 필요한 인수를 `setup` 메서드에 전달하므로 단위 테스트를 작성하기도 어렵다.

이보다 더 안 좋은 사례는 다음과 같이 구현된 생성자다.

```
public class Thing {
    private Weather currentWeather;
    public Thing(String weatherServiceHost) {
        currentWeather = getWeatherFromHost(weatherServiceHost);
    }
}
```

이 인스턴스를 생성하려면 인터넷 연결은 물론 다른 서비스가 필요하다. 다행히 이런 경우는 극히 드물다.

이런 패턴은 행위를 '캡슐화'해서 인스턴스를 더 쉽게 생성하려는 의도를 반영한 것이다. 하지만 내가 보기에 이 방법은 생성자와 소멸자(destructor)를 이용해 프로그래머가 리소스를 제어하는 C++에서나 사용하던 방법이다. 개별 클래스가 각각 필요한 내부 의존성을 관리할 수 있다면 상속 계층에서 클래스를 결합하는 것이 더 쉽다.

필자는 모듈라-3(Modula-3, https://oreil.ly/t2t4G) 경험에서 영감을 받은 방법을 선호한다. 즉, 생성자는 필드에 값을 대입하는 역할만 하는 것이다. 생성자가 해야 할 일은 올바른 인스턴스를 생성하는 것뿐이다. 객체 생성 시점에 수행해야 할 작업이 더 많다면 팩토리 메서드를 사용한다.

```
public class Thing {
    private final Fixed fixed;
    private final Details details;
    private NotFixed notFixed;

    public Thing(Fixed fixed, Details details, NotFixed notFixed) {
        this.fixed = fixed;
        this.details = details;
        this.notFixed = notFixed;
    }

    public static Thing forInternationalShipment(
            Fixed fixed,
            Dependencies dependencies,
            OtherStuff otherStuff) {
        final var intermediate = convertFrom(dependencies, otherStuff);
        return new Thing(fixed,
                         intermediate.details(),
                         intermediate.initialNotFixed());
    }
```

```
    public static Thing forLocalShipment(Fixed fixed,
                                         Dependencies dependencies) {
        return new Thing(fixed,
                         localShipmentDetails(dependencies),
                         NotFixed.DEFAULT_VALUE);
    }
}

final var internationalShipment =
    Thing.forInternationalShipment(fixed, dependencies, otherStuff);
final var localShipment = Thing.forLocalShipment(fixed, dependencies);
```

이 방법의 장점은 다음과 같다.

- 인스턴스 필드의 수명주기(life cycle)가 매우 명확하다.
- 객체 인스턴스의 생성과 사용이 분리되었다.
- 생성자와 달리 스스로의 역할을 명확히 표현하는 팩토리 메서드를 선언할 수 있다.
- 클래스와 인스턴스의 단위 테스트를 독립적으로 작성하기가 더 쉬워졌다.

이 방법에는 클래스 계층 구조상에서 생성자 논리를 공유할 수 없다는 단점도 존재한다. 하지만 클래스 계층 구조가 너무 깊어지는 것은 피해야 한다는 것에서 힌트를 얻어 헬퍼(helper) 메서드를 구현하면 얼마든지 극복할 수 있는 단점이다.

마지막으로 이 방법은 의존성 주입 프레임워크를 다루는 법에 주의해야 하는 이유이기도 하다. 객체의 생성이 복잡하고 리플렉션(reflection) 기반의 도구를 사용하기 쉽다는 이유로 필요한 매개변수를 모두 생성자에 몰아넣는 것은 다시 퇴보하는 것처럼 느껴지기 때문이다. 마찬가지로 '캡슐화' 때문에 비공개(private)로 선언한 필드에 리플렉션을 이용해(또는 생성자를 작성하지 않으려고) 직접 값을 대입하는 것은 타입 시스템을 망치는 길이며 단위 테스트를 더 어렵게 만들 뿐이다. 필드는 최소 기능 생성자를 이용해 값을 대입하는 것이 옳다. `@Inject`나 `@Autowired`는 주의해서 사용하고 모든 것을 명확히 표현하자.