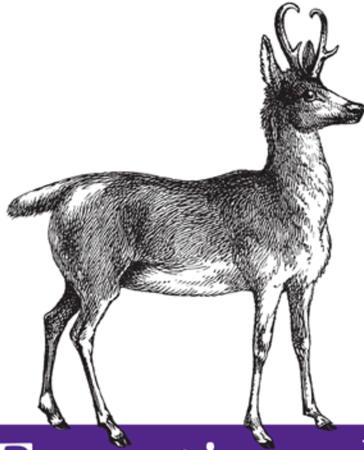
Hanbit eBook

Realtime 05

자바 개발자를 위한 함수형 프로그래밍

Functional Programming for Java Developers

딘 왐플러 지음 / **임백준** 옮김



이 도서는 O'REILLY의
Functional Programming for Java Developers의
번역서입니다.

자바 개발자를 위한

함수형 프로그래밍

지은이_**딘 왐플러**
Think Big Analytics 사에서 빅 데이터 관련 문제와 하둡 및 기계 학습 등을 전문적으로 다루는 컨설턴트다. 스칼라, JVM 에코시스템, 자바스크립트, 함수형 및 객체지향 프로그래밍, 애자일 방법론 등에도 정통하여, 관련 주제를 다루는 컨퍼런스의 주요 연사이기도 하다. 워싱턴 대학교에서 물리학 박사 학위를 받았다.

옮긴이_**임백준**
서울대학교에서 수학을 전공하고, 인디애나 주립대학에서 컴퓨터 사이언스를 공부했다. 삼성SDS, 뉴저지 소재 루슨트테크놀로지스에서 근무했고 지금은 월스트리트에 있는 회사에서 금융소프트웨어를 개발하고 있다. 뉴저지에서 아내와 두 딸과 함께 살고 있다. 『누워서 읽는 퍼즐북』(2010), 『프로그래밍은 상상이다』(2008), 『뉴욕의 프로그래머』(2007), 『소프트웨어 산책』(2005), 『나는 프로그래머다』(2004), 『누워서 읽는 알고리즘』(2003), 『행복한 프로그래밍』(2003, 이상 한빛미디어), 『프로그래머 그 다음 이야기』(공저, 2011, 로드북) 등의 도서를 집필하였으며, 『읽기 좋은 코드가 좋은 코드다』(2012, 한빛미디어)를 번역하였다.

자바 개발자를 위한 **함수형 프로그래밍**

초판발행 2012년 9월 19일

지은이 딘 왐플러 / **옮긴이** 임백준 / **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 978-89-7914-967-8 15560 / **정가** 11,000원

책임편집 배용석 / **기획** 김창수 / **편집** 김병희
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 이순옥
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 박주훈

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
Copyright ⓒ 2012 HANBIT Media, Inc.
Authorized Korean translation of the English edition of *Functional Programming for Java Developers, ISBN 9781449311032* ⓒ 2011 Dean Wampler. 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 저작권은 오라일리 사와 한빛미디어(주)에 있습니다.
저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

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.

저자 서문

자바 개발자를 위한 함수형 프로그래밍의 세계에 오신 걸 환영합니다.

자바 개발자가 함수형 프로그래밍FP(Functional Programming)을 배워야 하는 이유는 무엇일까요? 함수형 프로그래밍은 수십 년 동안 학문의 세계에서만 이야기된 것인데 우리에게 도움이 될까? 객체지향 프로그래밍OOP(Object-Oriented Programming)만으로는 충분하지 않다는 말인가? 이 책은 현재 우리가 마주하고 있는 문제를 해결하는데 있어서 함수형 프로그래밍이 도구로써 어떠한 역할을 하는지, 그리고 자바 개발자인 여러분이 함수형 프로그래밍을 어떻게 활용할 수 있는지에 대해서 설명합니다.

함수형 프로그래밍에 대한 최근의 관심은 병렬성parallelism을 통해 수평적인 확장을 구현하는 동시성concurrency 환경의 급속한 성장에서 비롯되었습니다. 멀티스레드 프로그래밍[Goetz2006 참고]을 제대로 구현하는 것은 매우 어렵기 때문에, 멀티스레드를 제대로 구현할 수 있는 개발자는 많지 않습니다. 앞으로 살펴보겠지만, 함수형 프로그래밍은 이와 같은 동시성을 지원하는 안정적인 소프트웨어를 작성하기 위한 보다 나은 방법을 제공합니다.

수평적인 확장은 다양한 관리와 분석을 요구하는 커다란 데이터 집합의 성장, 즉 빅 데이터로 일컬어지는 현상에 필요합니다. 이러한 데이터 집합은 전통적인 데이터베이스 관리 시스템이 감당하기에는 지나치게 크기 때문입니다. 이러한 수준의 데이터를 저장하고 처리하려면 컴퓨터 클러스터가 필요합니다. 오늘날에는 이렇게 커다란 데이터를 다루는 회사가 구글, 야후, 페이스북, 트위터 등으로 국한되어 있지는 않을 것입니다. 다른 조직 중에서도 이와 같은 문제에 직면하고 있는 곳은 많을 것입니다.

함수형 프로그래밍의 장점을 배우고 나면, 그러한 장점들이 자신이 작성하는 모든 코드의 질을 향상시킨다는 사실을 알게 될 것입니다. 몇 년 전에 함수형 프로그래밍을 처음 배웠을 때, 프로그래밍에 대한 나의 열정이 새롭게 자극되었습니다. 문제를 바라보는 새롭고 흥미로운 방법들을 알게 되었기 때문입니다. 함수형 프로그래밍의 엄밀함은 테스트 주도 개발방법론이 일반적인 설계와 테스트 과정에 기여하는 장점을 더욱 보강해 주었기 때문에, 내가 작성한 코드에 대해서 더 높은 확신을 가질 수 있었습니다.

나는 스칼라Scala 프로그래밍 언어[Scala 참고]를 통해서 함수형 프로그래밍을 공부했고, 알렉스 페인Alex Payne과 함께 [Programming Scala(O'Reilly)]라는 스칼라에 대한 책을 저술했습니다. 스칼라는 JVM 위에서 동작하는 언어로 자바 뒤를 잇는 잠정적인 후계자며, 객체지향과 함수형 프로그래밍을 하나로 묶는다는 목표를 가지고 있습니다. 클로저Clojure는 JVM 위에서 동작하는 언어로 알려진 또 하나의 함수형 프로그래밍 언어입니다. 이것은 리스프Lisp의 변종으로, 함수형 프로그래밍을 지원하기 위해서 객체지향 프로그래밍의 사용을 최소화합니다. 클로저는 앞으로 프로그래밍이 어떤 식으로 이루어져야 하는지에 대한 명확한 비전을 제시해 주는 언어입니다.

함수형 프로그래밍이 가지고 있는 많은 장점을 즐기기 위해서 반드시 새로운 언어를 배워야 할 필요가 없다는 사실은 다행스러운 일입니다. 1990년대 무렵 나는 C++를 사용하기 전까지 C로 작성하는 소프트웨어에서 객체지향 기법을 사용했습니다. 이와 마찬가지로 현재 자바와 같은 객체지향 프로그래밍 언어를 사용하고 있는 사람은, 함수형 프로그래밍의 여러 가지 아이디어를 자기가 사용하는 언어로 미리 적용해 볼 수 있습니다.

불행하게도 함수형 프로그래밍의 문법은 새로 배우는 사람에게 약간 어렵게 느껴질 것입니다. 이 책은 그런 사람에게 함수형 프로그래밍을 편하고 실용적으로 소개하는 것을 목적으로 합니다. 자바 개발자를 독자로 상정하고 있지만, 설명하는 원리는 일반적인 내용이기 때문에 객체지향 프로그래밍 언어를 다루는 사람이면 누구나 읽을 수 있습니다.

그렇지만 이 책을 읽기 위해서는 객체지향 프로그래밍에 익숙하고 자바 코드를 잘 이해할 수 있어야 합니다. 각 장의 뒤에는 방금 배운 내용을 연습하고 확장하는 데 도움을 주는 연습문제가 포함되어 있습니다.

이 책은 분량이 제한된 입문서입니다. 어떤 함수적 개념은 자바 언어를 이용해서 설명하기가 어렵기 때문에, 몇 가지 주제는 책의 완성도를 높이기 위해서 용어 정리 목록에는 포함시켰지만 본문에서는 다루지 않고 넘어갔습니다. 본문에 포함되지 않은 주제로는 커링currying, 부분 애플리케이션partial application, 컴프리헨션comprehension 등이 있습니다. 하지만 조합기combinator, 지연laziness, 모나드monad와 같은 주제에 대해서는 간단하게나마 설명할 것입니다. 함수형 프로그래밍을 처음 접하는 사람이라면 이러한 내용을 처음부터 자세하게 이해할 필요는 없습니다.

내가 그랬던 것처럼 여러분도 함수형 프로그래밍의 매혹에 사로잡히기를 기대해봅니다.

딘 왐플러

역자 서문

함수형 프로그래밍 언어는 객체지향 프로그래밍 프로그래밍 언어를 대체하는 존재가 아니라, 객체지향 프로그래밍 언어의 외연을 확장하고 내연을 풍부하게 만들어주는 도우미다. 객체지향이라는 나무에 함수형 프로그래밍 패러다임이라는 비료를 뿌려주면 간결하고, 효율적이며, 병렬적인 아름다운 프로그램이라는 꽃이 활짝 피어난다. 함수형 프로그래밍의 장점에 대해서는 이 책의 저자 서문에 잘 설명되어 있으므로 중복하지 않겠지만, 자바나 C#과 같은 기존의 객체지향 프로그래밍 언어가 함수형 프로그래밍의 장점을 적극적으로 받아들이고 있는 현상이나, 스칼라나 F# 같은 함수형 프로그래밍 언어가 많은 주목을 받고 있는 최근의 현상은 범상치 않다.

2013년 하반기에 출시될 것으로 예상되는 자바 8은 클로저나 람다 같은 함수형 프로그래밍 언어의 기능을 포함할 것이라고 이야기되고 있다. 매우 늦은 감이 있지만 이제라도 이러한 기능이 자바에 포함된다면 다행이다. 클로저나 람다가 개발자의 일상생활에 가져다주는 효과를 이해하지 못하는 자바 개발자도 많을 것이다. 이런 문법이 없어도 자바로 구현할 수 있는 기능에는 아무런 제한이 없기 때문이다(실제로 제한은 없다. 하지만 이런 식으로 따지면 어셈블리 언어로 구현할 수 있는 기능에도 제한이 없기는 마찬가지다). 그런 사람도 일단 이러한 문법적 기능을 사용해서 코딩하기 시작하면 인터페이스와 익명 클래스를 사용하던 과거로 돌아가기 어려울 것이다. 추상은 일방통행이기 때문이다. 추상은 언제나 더 간결하고 더 우아한 코드를 가능하게 만드는 방향으로만 전진한다.

왐플러의 책은 짧지만 함수형 프로그래밍 기법에 익숙하지 않은 사람에게는 결코 만만한 책이 아니다. 나는 1996년 이래로 지금까지 자바를 사용하고 있는 자바 개발자지만, 수년 전부터 C# 언어와 F# 언어를 통해서 함수형 프로그래밍의 개념을 익

혔다. 그렇기 때문에 이 책을 번역하는 과정에서 왐플러가 의도적으로 설명을 초보적인 수준에서 멈추고 있다는 사실을 확인할 수 있었다. 함수형 프로그래밍의 기법이 갖는 강력한 힘과 장점이 매우 압축적인 책의 분량 때문에 충분히 납득할 수 있는 방식으로 제시되지 않고 있는 것이다. 그렇기 때문에 이 책을 통해서 함수형 프로그래밍과 관련된 많은 것을 알게 되기는 어려울 것이다. 하지만 처음부터 쉽게 읽히지 않는 함수형 프로그래밍의 개념을 자세하게 다루는 책을 읽기보다 자바 개발자를 위한 기초적인 개념을 소개하는 이 책을 먼저 읽고 다른 책을 읽으면 도움이 되리라고 생각한다.

함수형 프로그래밍을 공부하지 않는다고 해도 개발자로서의 일상생활에는 아무런 지장이 없다. 하지만 함수형 프로그래밍 기법을 익힌 개발자와 그렇지 않은 개발자가 작성하는 코드의 품질은 완전히 다르다. 미대 지망생의 붓질과 램브란트의 붓질만큼이나 차이가 날 것이다. 이것은 나의 개인적인 경험을 통해서 증언할 수 있는 사실이며, 왐플러 역시 책의 본문에서 똑같은 말을 하고 있다. 지금 나의 역자 서문을 읽고 있는 독자라면 이미 간결하고 우아한 프로그래밍에 많은 관심을 가지고 있는 지적욕구가 왕성한 개발자일 것이다. 망설일 이유가 없다. 왐플러의 글을 읽는 것을 시작으로 함수형 프로그래밍이라는 새로운 패러다임을 향한 긴 여정의 첫걸음을 떼기 바란다. 어느 정도 시간이 흐른 후에 뒤를 돌아보면, 지금 용감하게 첫걸음을 뗀 자기 자신을 매우 자랑스럽게 생각하게 될 것이다.

2012. 9. 1. 뉴저지에서 임백준

예제 파일

예제 파일은 http://examples.oreilly.com/9781449311032/에서 받을 수 있습니다.

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 준비 중이며, 조만간 선보일 예정입니다.

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.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 계실 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전되지 않습니다.

차례

01 왜 함수형 프로그래밍을 배워야 하는가? 01

1.1 동시성 프로그램을 잘 작성해야 한다 03
1.2 문제는 데이터를 관리하는 문제로 귀결된다 03
1.3 함수형 프로그래밍은 모듈화되어 있다 05
1.4 더 빠르게 작업해야 한다 07
1.5 함수형 프로그래밍은 단순함으로의 복귀다 08

02 함수형 프로그래밍이란 무엇인가? 09

2.1 함수형 프로그래밍의 기본 원리들 10
2.1.1 변경 가능한 상태 피하기 10
2.1.2 1등 시민으로서의 함수 14
2.1.3 람다와 클로저 18
2.1.4 고계함수 20
2.1.5 부수효과가 없는 함수 20
2.1.6 재귀 21
2.1.7 게으른 vs 부지런한 계산 23
2.1.8 선언적 vs 명령형 프로그래밍 25
2.2 타입 설계하기 27
2.2.1 null을 어떻게 해야 하는가 27
2.2.2 대수적 데이터 타입과 추상적 데이터 타입 34
2.3 연습문제 36

03 데이터 구조와 알고리즘 37

3.1 리스트 37
3.2 맵 47
3.3 조합기 함수: 컬렉션 파워 툴 48
3.4 영속적인 데이터 구조 60
3.5 데이터 구조와 알고리즘에 대한 몇 가지 추가적인 고찰 62
3.6 연습문제 65

04 함수적 동시성 67

4.1 액터 모델 67
4.2 소프트웨어 트랜잭션 메모리 72
4.3 연습문제 77

05 더 나은 객체지향 프로그래밍 78

5.1 명령적인, 변경 가능한 코드 78
5.2 리스코프 치환 원칙 79
5.3 설계 패턴에 대하여 81
5.3.1 패턴 매칭 82
5.4 좋은 타입을 구성하는 요소들 85
5.5 객체지향 미들웨어에 대해 다시 생각하기 87
5.6 연습문제 88

06 이제는 어디로 갈 것인가 89

6.1 자바를 위한 함수적 도구들 ... 91
6.2 복습 ... 91
6.3 연습문제 ... 94

부록 Appendix 95

참고자료 ... 95
용어정리 ... 102

1 | 왜 함수형 프로그래밍을 배워야 하는가?

몇 년 전에 많은 개발자가 동시성 문제에 접근하기 위한 최선의 방법이 함수형 프로그래밍이라고 이야기했을 때, 스스로 공부해서 그런 주장의 진위를 직접 확인하겠다고 마음을 먹었다. 그리고 함수형 프로그래밍 학습을 통해서 뭔가 새로운 아이디어를 얻을 수 있긴 하겠지만, 내가 주로 사용하는 언어는 여전히 객체지향 프로그래밍 언어일 것이라 생각했다. 그런데 이러한 생각은 잘못된 것이었다.

함수형 프로그래밍을 학습하면서, 원래 기대했던 바와 같이 동시성 문제에 접근하는 새로운 방법들을 알게 되었다. 또한, 이전에는 알지 못했던 명확한 방식으로 타입이나 함수를 설계하는 방법도 알게 되었다. 전보다 더 간결한 코드를 작성하는 것도 가능해졌다. 함수형 프로그래밍은 모듈의 경계를 어디쯤으로 정해야 하는지, 모듈의 재사용 가능성을 어떻게 더 높일 수 있는지에 대해서 이전과 다르게 생각할 수 있도록 도와주었다. 함수형 프로그래밍 커뮤니티가 혁신적이고 더 강력한 타입 시스템을 구축함으로써, 프로그램의 정확성을 향상시키고 있다는 사실도 발견하였다. 엄청난 분량의 데이터 집합을 다루어야 하고, 빠르게 변화하는 요구사항에 민첩하게 적응해야 하며, 촉박한 스케줄을 감수해야 하는 현대 개발자의 고유한 문제들을 해결하는 데 있어서 함수형 프로그래밍이 매우 유용하다는 결론을 내리게 되었다.

나는 객체지향 프로그램 개발자이면서 함수형 프로그램을 필요한 곳에 양념처럼 뿌리는 사람이 아니라, 객체를 제한적으로 사용하면서 기본적으로 함수형 프로그램을 작성하는 사람이 되었다. 다시 말해서, 처음에는 동시성 문제에 대한 연구를 목적으로 함수형 프로그램에 입문했는데 결과적으로는 완벽한 패러다임의 변화를 경험하게 된 것이다.

재미있는 사실은 우리가 이미 이러한 현상을 이전에도 경험한 바가 있다는 것이다.

80년대에 객체지향 프로그램이 메인스트림mainstream이 되어가고 있을 때, 지금과 똑같은 현상이 존재했다. 객체는 그래픽 관련 컴포넌트를 나타내기에 매우 적합했기 때문에, 그래픽 사용자 인터페이스GUI를 개발하는 사람들은 자연스럽게 객체지향 프로그램을 사용하였다. 그리고 객체를 사용하기 시작하면서, 객체가 다른 여러 가지 도메인을 표현하는 것도 적합하다는 사실을 깨닫기 시작했다. 어떤 도메인을 일단 객체로 모델링하고 그 모델을 곧바로 코드로 옮길 수 있었던 것이다. 다양한 형태의 입출력처럼 매우 자세한 구현조차 객체를 사용하면 더 쉽게 모델링할 수 있는 것처럼 보였다.

그렇지만 한 가지 분명히 짚고 넘어갈 것이 있다. 함수형 프로그래밍과 객체지향 프로그래밍은 어느 것이든 만병통치약이 아니라 단지 도구일 뿐이라는 사실이다. 두 프로그래밍 방법은 장점과 단점을 동시에 가지고 있다. 일단 어느 하나를 사용하기 시작하면, 심지어 다른 방법이 더 나을 때조차 항상 자기가 사용한 것만 계속 사용하는 관성이 있다. 어쨌든 오랫동안 자기가 맡은 역할을 훌륭하게 수행해온 객체가 오늘날에 이르러서 이전보다 가치가 없게 되었다는 사실은 받아들이기 어렵다. 그렇지 않은가? 함수형 프로그래밍에 더 많은 관심을 갖는 것이, 여전히 나름의 가치를 지니고 있는 객체를 곧 포기한다는 것을 의미하지는 않는다. 오히려 오늘날 우리가 직면하고 있는 프로그래밍과 관련된 문제들 때문에 객체가 처음부터 가지고 있었던 단점이 부각되고 있는 상황이라고 말하는 것이 옳을 것이다. 수십 년 전에 객체가 처음 부상하던 시절과 지금은 확실히 다르기 때문이다.

지금까지 간략하게 내가 어떻게 함수형 개발자가 되었는지, 그리고 여러분이 함수형 프로그래밍을 학습해야 하는 이유가 무엇인지에 대해서 살펴보았다. 함수형 프로그래밍은 이어서 설명할 문제를 해결하고자 할 때 최선의 방법을 제공해준다. 이러한 문제는 내가 실제로 매일 부딪히는 문제들이다.

1.1 동시성 프로그램을 잘 작성해야 한다

여러 스레드에 공유되어 있으면서 상태를 변경할 수 있는 데이터가 있다면, 그것은 매우 신중하게 동기화되어야 한다. 지금까지 이러한 데이터에 대한 멀티스레딩 코드를 작성하는 일은 소수의 똑똑한 친구들이 전담하는 것이 일반적이었다. 어떤 사람은 현장에서 발견된 동시성과 관련된 까다로운 버그 때문에 한밤중에 전화를 받은 괴로운 경험이 있을 것이다. 그렇지만 그런 예외적인 경우를 제외하면 대부분의 개발자는 동시성과 관련된 문제를 무시하고 살아갈 수 있었다.

하지만 오늘날에는 간단한 전화기조차 여러 개의 CPU 코어를 가지고 있다(아직 그렇지 않다면 다음번에 구매하는 전화기는 분명히 그럴 것이다). 안정적인 동시성 소프트웨어를 작성하는 것은 더 이상 선택사항이 아니다. 이런 상황에서 함수형 프로그래밍이 동시성에 대한 명확한 원리를 제공하고, 우리의 작업이 쉽게 진행될 수 있도록 도와주는 높은 수준의 추상적 개념을 탄생시켰다는 사실은 다행이다.

NOTE_ 공유할 수 있으며, 값이 변하는 상태에 대한 동기화를 요구하는 멀티스레드 프로그래밍은 동시성에 대한 일종의 어셈블리어에 해당한다.

1.2 문제는 데이터를 관리하는 문제로 귀결된다

최근 들어 나는 맵리듀스MapReduce [Hadoop 참고]를 이용해서 구현된 아파치 하둡Apache Hadoop 환경에서 대용량 데이터를 다루는 작업을 수행하는 일이 많아졌다. 매일 테라바이트terabytes 정도의 데이터가 새로 입력되거나 페타바이트petabytes 정도의 데이터를 분석하고 저장해야 한다면, 객체를 둘러싸고 있는 오버헤드를 감당하기가 어렵다. 이러한 경우에는 오버헤드가 최소화되어 있는 대단히 효율적인 데이터 구조와 데이터 연산이 필요하다. 애자일 진영에서 이야기하는 "가장 간단한 방법은 무엇인가?"라는 과거의 질문은 이러한 맥락에서 새로운 의미를 얻게 되었다.

데이터베이스를 사용하는 전형적인 IT 애플리케이션이 작은 데이터 집합을 어떻게 관리하는지 생각해보자. 커다란 데이터를 다루어야 하는 상황에서 객체가 과도한 오버헤드라는 문제를 제기한다면, 작은 데이터를 다루는 상황에서 그러한 오버헤드는 어떤 문제를 일으킬까? 사실 이런 경우에는 객체의 오버헤드가 애플리케이션 성능이나 데이터 저장과 관련해서 별다른 문제를 일으키지 않는다. 하지만 팀이 아주 민첩하게 움직여야 한다면 문제를 일으킬 가능성이 있다. 작은 규모의 팀이 해마다 IT 애플리케이션의 성능을 향상시켜야 한다면 어떻게 민첩함을 유지할 수 있을까? 코드베이스[01]를 어떻게 최대한 간결하게 유지할 수 있는 것일까?

이런 문제를 고민하는 과정에서 도메인 객체 모델을 코드 안에 우직하게 구현하는 방식에 대해서 의문을 품게 되었다. 객체관계 매핑ORM(Object-Relational Mapping)이나 그와 유사한 다른 미들웨어는 관계형 데이터를 객체로 변환하고, 객체를 애플리케이션 주위로 전송하여 서로 주고받으며, 최종적인 업데이트를 위해서 객체를 다시 관계형 데이터로 변환시키는 일련의 과정을 통해서 지속적으로 오버헤드를 발생시킨다. 오버헤드를 발생시키는 추가적인 코드는 테스트하고 유지보수하는 것이 당연한 일이다.

이러한 관행이 생겨난 이유는 우리가 객체를 지나치게 사랑하지만, 관계형 데이터를 미워하기 때문일 것이다. 어쩌면 관계형 데이터베이스를 이용해서 작업하는 것을 원치 않는 것일 수도 있다(개인적인 경험을 토대로 이야기하고 있는 것일 뿐이다). 하지만 SQL 질의문을 통해서 얻은 결과 같은 관계형 데이터는 함수적인 방법을 통해서 관리할 수 있는 데이터 컬렉션에 불과하다. 그렇다면 객체를 경유하는 대신 그러한 데이터 위에서 직접 작업을 하는 것이 더 낫지 않을까?

기본적인 데이터 컬렉션 위에서 직접 작업하는 것이 객체 모델을 이용해서 작업을 하는 것보다 오버헤드를 적게 발생시키고, 객체 모델과 마찬가지로 중복된 코드를

---

01 (역자주) 프로젝트에 사용된 소스코드의 집합을 의미한다.

지양하며, 코드의 재사용도 촉진한다는 사실을 이 책에서 보여줄 것이다.

1.3 함수형 프로그래밍은 모듈화되어 있다

내가 경험한 어떤 클라이언트는 수년 전에 터무니없이 부풀어 오른 코드베이스 때문에 애를 먹고 있었다. 그들은 경쟁업체들과 싸움을 벌이고 있는 중이었다. 어느 날, 문제의 본질을 담고 있는 대상을 발견했다. 우연히 책상과 책상을 가르고 있는 길이 1.5 미터 정도의 칸막이에 UML 다이어그램이 붙어 있는 것을 보았는데, 거기에 있던 Customer라는 이름의 클래스를 각별히 기억하게 되었다. 그 클래스는 1.5 미터 벽의 끝에서 끝까지 연결되어 있었다. 이것은 모듈화의 완벽한 실패다. 적절한 수준의 추상화abstraction와 분해decomposition를 포착하는 데 실패한 것이다. Customer 클래스는 고객과 조금이라도 관련이 있는 모든 잡동사니를 한 곳에 담아놓는 쓰레기통 역할을 하고 있었다.

80년대에 객체지향 프로그래밍이 주목을 받던 무렵에, 사람들은 객체가 재사용 가능한 컴포넌트를 만들어내는 문제에 대한 최종적인 해답을 제공해줄 것이라고 기대했다. 그렇게 된다면 앞으로는 컴포넌트를 조립해서 애플리케이션을 만들어낼 수 있으며, 개발과 관련된 시간과 비용은 획기적으로 줄어들 것이라고 기대한 것이다. 이러한 비전은 너무나 당연한 것으로 보였기 때문에, 기대한 대로 상황이 전개되지 않았다는 사실을 인식하기가 오히려 쉽지 않을 정도였다.

재사용 가능한 라이브러리를 성공적으로 제공한 사례 대부분은 자기만의 고유한 표준을 제정하고 다른 사람들에게 자신의 표준을 따르도록 강제한 플랫폼이었다. 이러한 예는 JDK, 스프링 프레임워크, 이클립스의 플러그인 API 등을 포함한다. 심지어 Apache Commons와 같은 대부분의 제3자 컴포넌트 라이브러리들도 반드시 따라야 하는 자기만의 고유 API를 정하고 있다. 우리가 필요로 하는 API 이외의 코드는 프로젝트를 수행할 때마다 스스로 다시 작성해야 한다. 결국 객체지향 소프트웨어 개발은 우리가 한때 희망했던 것처럼 컴포넌트 어셈블리의 역할을 수

행한 것이 아니다.

거의 무제한에 가까운 객체의 유연성은 오히려 코드의 재사용 가능성을 침해한다. 객체가 상호작용하는 방식에 대한 표준이 없고, 심지어 객체의 이름을 정하는 것에 대해서조차 의견일치를 볼 수 없기 때문이다. 더 많은 제한을 가지고 있는 시스템이 더 잘 모듈화되어 있다는 사실은 역설적이다. "설계 규칙: 모듈화의 힘(Design Rules: The Power of Modularity)"[Baldwind2000 참고]는 PC 산업의 폭발적인 성장이, IBM이 개인용 컴퓨터 하드웨어 아키텍처에 대한 실질적인 표준을 제정하는 것을 가능하게 해주었다고 하였다. 주변장치나 연결장치를 위한 버스가 표준화됨에 따라서 다른 회사가 새롭고, 향상된 기능에 값은 싼 드라이브, 마우스, 모니터, 마더보드 등을 제작할 수 있게 되었던 것이다. 디지털 전자제품 자체도 모듈화 시스템에 대한 훌륭한 예다. 각 전선은 0 아니면 1이라는 신호를 전달한다. 하지만 전선을 8개, 16개, 32개, 64개씩 묶으면 컴퓨터에서 보는 바와 같이 온갖 종류의 환상적인 결과를 낳을 수 있게 된다.

객체지향 기반의 컴포넌트에서는 이와 비슷한 종류의 표준화가 존재하지 않는다. CORBA나 COM 같은 다양한 시도들은 성공을 거두는 것처럼 잠시 보이기도 했지만, 결국에 가서는 객체라는 것이 적절한 수준의 추상단위가 아니라는 사실 때문에 실패하고 말았다. 고객customer이라는 개념은 별로 새로운 것이 아니지만, 프로젝트를 수행할 때마다 새로운 모습의 클래스를 정의하는 일을 멈출 도리가 없다. 각 프로젝트는 고유의 문맥과 요구사항을 가지고 있기 때문이다.

그렇지만 객체라는 것이 데이터 집합에 불과하다는 사실을 깨달으면, 객체보다 더 낮은 수준에서 오히려 더 나은 표준적인 추상화를, 예를 들어, 앞에서 살펴본 전자회로와 같은 추상화 도구를 찾는 방법을 알 수 있게 될 것이다. 이러한 표준은 리스트list, 맵map, 셋set과 같은 근본적인 컬렉션, 혹은 숫자 값(금융 애플리케이션에서 사용하는 Money와 같은)이나 몇몇 잘 정의된 도메인 개념들을 모두 포함한다.

모듈화를 위해서 필요한 것은 함수형 프로그래밍에서 말하는 함수의 성질이다. 그것은 부수효과side effects를 갖지 않고, 다른 객체에 종속되지 않기 때문에 많은 곳에서 재사용할 수 있다. 결론적으로 함수형 프로그래밍은 더 유용하고, 재사용이 더 편리하고, 구성compose이 용이하고, 테스트하기 더 간편한 추상화를 제공한다.

NOTE_ 어떤 임의의 복잡한 객체(원시 값primitive values과 같은)는 원자적atomic 값과 그러한 값들을 포함하는 데이터 컬렉션으로 분해할 수 있다.

1.4 더 빠르게 작업해야 한다

개발 사이클은 점점 0으로 수렴하고 있다. 여러분이 나처럼 프로젝트가 보통 몇 달씩 혹은 몇 년씩 걸리던 시절에 직업적인 프로그래밍을 시작한 사람이라면 이러한 말이 정신 나간 소리로 들릴 것이다. 하지만 오늘날 인터넷 사이트 중에는 하루에만 몇 번씩 코드를 전개deploy하는 곳이 많다. 우리는 모두 작업을 더 빨리, 물론 품질은 손상하지 않으면서 수행해야 한다는 압력에 시달리고 있다.

스케줄이 길던 시절에는 도메인을 신중하게 모델링한 다음, 그 모델을 코드로 작성하는 것이 의미가 있었다. 만약 실수를 저지르면, 그것을 수정하기 위해서 몇 달 뒤에 출시되는 릴리즈까지 기다려야 했다. 오늘날에는 대부분 프로젝트에서 도메인의 내용을 명확하게 이해하는 것이 무언가를 빠르게 출시하는 것보다 중요하지 않게 되었다. 도메인에 대해서 우리가 이해하고 있는 내용은 새로 출시된 소프트웨어를 사용한 사용자의 머릿속에 뭔가 새로운 통찰이 생겨남에 따라서 순식간에 변하기도 한다. 도메인의 어떤 부분을 잘못 이해했다면, 자주 수행되는 전개를 통해서 문제를 빠르게 수정할 수도 있다.

신중한 모델링이 전보다 중요하지 않게 되었다면, 객체 모델을 우직하게 구현하는 것이 전보다 더 의심스러운 방식이 된 것도 사실일 것이다. 애자일 소프트웨어 개

발이 품질과 변화에 대한 응답속도를 개선한 것은 틀림없지만, 코드가 사용자 요구사항에 대해서 최소한으로 충분한 수준을 유지하면서 동시에 변화에 대해서는 유연할 수 있게 만드는 방법을 다시 생각할 필요가 있다. 함수형 프로그래밍은 바로 이 지점에 도움을 준다.

1.5 함수형 프로그래밍은 단순함으로의 복귀다

지금까지 앞에서 살펴본 논점을 점검하자면, 함수형 프로그래밍은 우연히 발생한 복잡성accidental complexity에 대한 응답이라고 생각한다. 우연히 발생한 복잡성이란 문제의 도메인 자체가 본질적으로 품고 있는 복잡성이 아니라 도메인을 구현하기 위해서 선택된 방식에서 발생하는 복잡성을 의미한다. 예를 들어서, 내가 보기에 오늘날 애플리케이션 대부분에서 사용되고 있는 객체지향 미들웨어는 불필요하고 낭비적이다.

이러한 주장이 일부 도발적이라는 사실을 알고 있다. 여러분에게 함수형 프로그래밍 당원이 되기 위해서 객체를 완전히 포기하라고 주문하는 것은 아니다. 다만 전보다 더 큰 도구상자와 더 확장된 시야를 제공해 주려는 것뿐이다. 따라서 여러분이 설계와 관련된 선택을 내릴 때 전보다 더 현명한 선택을 할 수 있고, 소프트웨어 개발과 관련된 기술과 과학에 대한 뜨거운 열정을 다시 한 번 품게 되기를 희망할 뿐이다. 지금까지 살펴본 이 짧은 소개가 여러분에게 내가 생각을 바꾼 이유를 설명해 주었기를 바란다. 어쩌면 앞으로 당신의 생각도 바뀔지 모르는 일이 아닌가?

자, 시작해보자!

2 | 함수형 프로그래밍이란 무엇인가?

순수한 의미에서 보았을 때 함수형 프로그래밍은 함수, 변수, 값들이 수학에서 어떻게 작동하는지에 근거를 두고 있다. 수학에서 함수, 변수, 값들이 나타나는 모습은 대부분의 프로그래밍 언어에서 나타나는 모습과 다르다.

함수형 프로그래밍은 디지털 컴퓨터가 존재하기 전에 이미 시작되었다. 계산방법computation의 이론적 토대는 대개 1930년대에 알론조 처치Alonzo Church와 해스켈 커리Haskell Curry가 개발하였다.

알론조 처치는 함수를 정의하고 실행하는 방식(함수를 적용한다고 말하기도 한다)을 설명하는 람다 계산법Lambda Calculus을 개발했다. 오늘날 대부분의 프로그래밍 언어가 사용하는 문법은 바로 이 모델을 반영한 것이다.

(해스켈 언어의 이름이 유래된) 해스켈 커리는 계산에 대한 대안적인 이론적 근거를 제공하는 조합논리Combinatory Logic를 개발하는 데 도움을 주었다. 조합논리는 기본적으로 함수에 해당하는 조합기combinator가 하나의 계산 과정을 나타내기 위해서 어떻게 조합을 수행하는지 검사한다. 이러한 조합논리를 유용하게 사용한 예가 해석기parser를 구성하기 위한 빌딩블록이다. 그리고 전체적으로 계획되어 있는 계산의 각 단계를 나타낼 때 유용하다. 각 단계에서 버그의 유무를 조사하거나 최적화 가능성을 검사할 때 사용된다.

최근에 와서는 범주이론Category Theory은 IOInput and Output(입력과 출력)와 같은 부수효과를 코드의 본체로부터 완벽하게 분리해서 어떠한 부수효과도 갖지 않도록 계산을 구축하는 방법을 제공해 주었다. 이러한 방법은 함수형 프로그래밍에 필요한 아이디어의 원천이 되었다.

함수형 프로그래밍에 내재된 문법의 많은 부분이 수학에 근거를 두고 있기 때문에,

수학과 관련된 경험이 많지 않은 사람은 이러한 문법이 어렵게 느껴질 수도 있다. 반면, 객체지향은 좀 더 일상적인 직관에 가까워 접근이 용이한 것처럼 느껴진다. 그렇지만 다행스럽게도 수학과 관련된 내용을 자세하게 공부하지 않아도 함수형 프로그래밍의 원리를 배우고 사용하는 데에는 문제가 없다.

함수형 프로그래밍의 아이디어를 처음으로 활용한 언어는 리스프Lisp다. 리스프는 1950년대 후반에 개발되었는데, 포트란Fortran 다음으로 세계에서 두 번째로 오래된 프로그래밍 언어에 해당한다. 캐멀Caml, 오캐멀OCaml(객체지향 하이브리드 언어), 마이크로소프트의 F# 등을 포함하는 ML 계열의 프로그래밍 언어는 1970년대에 시작되었다. 함수형 언어의 가장 순수한 의미에 근접한 언어로 알려져 있는 것은 1990년대 초반에 시작된 해스켈Haskell일 것이다. 최근에 등장한 다른 함수형 언어로는 클로저Clojure와 스칼라Scala가 있는데, 둘 다 JVM 환경에서 동작하며 .NET 환경으로 이식이 진행되고 있다. 오늘날에는 많은 언어가 함수형 프로그래밍에서 빌어온 아이디어를 사용하고 있다.

2.1 함수형 프로그래밍의 기본 원리들

모든 프로그래밍 언어가 어쨌든 함수를 포함하고 있지 않은가? 만약 그렇다면 어째서 그런 프로그래밍 언어들을 함수형 언어로 부르지 않는 것일까? 함수형 프로그래밍 언어들은 함수 자체 말고도 몇 가지 기본적인 원리를 포함하고 있기 때문이다.

2.1.1 변경 가능한 상태 피하기

첫 번째 원리는 변경 불가능한 값immutable value을 이용하는 것이다. 학교에서 배웠던 유명한 피타고라스의 정리를 기억해보자.

$x^2+y^2=z^2$

x와 y라는 변수에 x=3, y=4라는 값을 대입하면 z의 값을 계산할 수 있다(이 경우에 z의 값은 5다). 여기에서 핵심적인 아이디어는 계산을 수행하는 동안 이러한 값들이 절대로 변하지 않는다는 사실이다. 이러한 수학공식에서는 3++와 같은 표현은 아무런 의미가 없다. 어느 변수에 새로운 값을 설정할 수 있을 뿐이며 일단 값이 설정되면 그 값을 바꿀 수 없다.

대부분의 프로그래밍 언어는 어떤 값과(메모리상의 어떤 상수와) 그 값을 가리키는 변수를 엄격하게 구분하지 않는다. 자바는 값을 변경하는 할당을 방지하기 위해서 final이라는 키워드를 사용한다. final을 사용하면 수학공식에서 사용하는 변수들처럼 변경이 불가능한 값이 된다.

값이 변경되는 것을 피해야 하는 이유는 무엇일까? 값이 변경되는 것을 허용하면 멀티스레드 프로그래밍이 어려워진다. 여러 개의 스레드가 공유하고 있는 값이 변경될 수 있다면, 그 값에 대한 접근이 동기화되어야만 한다. 그렇지만 이러한 동기화는 전문가들조차 제대로 구현하기 어려운 매우 까다롭고 실수하기 쉬운 과정이다[Goetz2006 참고]. 하지만 값을 변경 불가능하게 만들면, 동기화와 관련된 문제는 깨끗이 사라진다. 여러 개의 스레드가 하나의 값을 동시에 읽어도 아무런 문제를 일으키지 않으므로, 멀티스레드 프로그래밍이 훨씬 쉬워진다.

변경 불가능한 값이 갖는 두 번째 장점은 프로그램의 정확성을 높여준다는 것이다. 변경 가능한 값들, 특히 값의 변화가 한 곳에서 이루어지지 않고 여러 곳에 흩어져 있으면 코드의 흐름을 이해하거나 테스트하기 어렵다. 큰 시스템에서 흔히 발견되는 가장 수정하기 어려운 버그는 어떤 상태의 변경이 예측 불가능한 임의의 장소에서, 즉 프로그램의 바깥에 존재하는 클라이언트 코드에 있는 경우다.

고객들의 주문을 저장하는, 변경이 가능한 리스트를 사용하는 다음 예를 살펴보자.

```
public class Customer {
    // setter 메서드가 없다
    private final List<Order> orders;
    public List<Order> getOrders() { return orders; }
    public Customer(...) {...}
}
```

Customer의 클라이언트가 Orders 리스트의 내용을 보고 싶어하는 것은 이상한 일이 아니다. 하지만 getOrders 메서드처럼 리스트를 코드 바깥에 완전히 드러내는 것은 리스트에 대한 통제권을 완전히 상실한다는 것을 의미한다. 이렇게 되면 클라이언트는 우리가 모르는 사이에 리스트 내용을 변경할 수도 있다. orders라는 변수를 final로 선언했고 그에 대한 setter 메서드도 제공하지 않았지만, 어떤 새로운 리스트를 orders라는 변수에 할당하는 것만 막을 수 있을 뿐이다. 리스트의 내용 자체가 변경되는 것은 막을 수 없다.

getOrders가 리스트의 복사본을 리턴하도록 만들거나 orders에 대해서 일정하게 통제된 접근만 허용하는 별도의 메서드를 Customer에 더함으로써 이러한 문제를 피할 수 있다. 하지만 리스트를 복사하는 것은, 예컨대 리스트의 크기가 클 때는 비용이 따르는 작업이다. 또한, 접근을 통제하기 위해서 특별한 방식으로 구현된 메서드를 더하는 것은 객체의 복잡성을 증가시키고, 테스트를 어렵게 만들며, 다른 개발자가 해당 클래스의 내용을 이해하기 어렵게 만든다.

만약 주문을 담고 있는 리스트 자체가 변경 불가능한 것은 물론, 그 안에 담긴 요소들도 변경이 불가능하다면 이런 걱정은 할 필요가 없다. 클라이언트 코드는 getter 메서드를 통해서 주문을 읽을 수 있지만 아무것도 변경할 수 없다. 이렇게 할 수만 있으면 객체의 상태에 대해서 통제권을 완전히 장악할 수 있다.

주문을 담은 리스트의 내용이 변경될 때, 리스트의 크기가 엄청나게 크다면 어떻게 할 것인가? 이렇게 커다란 리스트의 복사본을 만드는 과정이 지나친 오버헤드를 발생시킨다면 차라리 리스트를 변경 가능하게 만드는 것이 낫지 않을까? 다행히도 우리는 커다란 데이터 구조를 매우 효율적으로 복사하는 방법을 이미 알고 있다. 변경할 필요가 없는 대부분의 요소를 재사용하는 것이다. 주문 리스트에 새로운 주문을 더하는 경우를 생각해보자. 이런 경우에는 리스트의 나머지 부분을 모두 재사용할 수 있다. 이러한 방법은 3장에서 살펴볼 것이다.

변경 가능한 값을 완전히 피할 수는 없다. 모든 프로그램은 어떤 식으로든 (모종의 변경을 초래하는) IO를 수행해야 한다. 그렇지 않다면 프로그램은, 유명한 농담처럼 단지 CPU의 온도를 높이는 일 말고는 하는 것이 없게 된다. 함수형 프로그래밍은 변경 가능한 값을 언제, 어디서, 어떻게 이용할 것인지에 대해서 신중하게 생각하도록 도와준다. 변경 가능한 값들을 잘 정의되어 있는 영역에 존재하도록 몰아넣고 코드의 나머지 부분은 변경되는 값들로부터 완전히 자유롭게 만든다면, 코드의 안정성과 모듈성은 크게 향상된다.

이렇게 정해진 영역에서만 일어나는 값의 변경도 여전히 멀티스레딩 환경에서 안전하게 수행되어야 한다. 소프트웨어 트랜잭션 메모리Software Transaction Memory와 액터 모델Actor Model은 이와 같은 종류의 안전성을 제공한다. 이런 방법은 4장에서 살펴볼 것이다.

NOTE_ 객체를 변경 불가능하게 만들어라. 필드는 final로 선언하라. 필드에 대한 getter 메서드는 반드시 필요한 경우에만 제공하라. final로 선언된 객체들의 상태가 변화할 수 있다는 사실에 유의하라. 변경이 가능한 컬렉션은 신중하게 사용하라. [Bloch2008]의 “변경 가능성을 최소화하라” 섹션에서 더 많은 조언을 확인할 수 있다.

2.1.2 1등 시민으로서의 함수

자바에서는 객체와 원시 값들을 메서드에 전달하고, 그 값을 리턴받고, 리턴받은 값을 다른 변수에 할당하는 데 익숙하다. 이것은 객체와 원시 값들이 자바에서는 1등 시민으로 대접받는다는 사실을 의미한다. 비록 리플렉션reflection API가 클래스 자체에 대한 정보를 제공하기는 하지만 클래스는 1등 시민이 아니라는 사실에 유의하기 바란다. 자바에서 함수는 당연히 1등 시민이 아니다. 잠시 함수와 메서드의 차이에 대해서 짚고 넘어가도록 하자.

> **NOTE_** 메서드는 어떤 특정한 클래스에 소속된 코드의 블록이다. 만약 static으로 정의되었다면 해당 클래스의 문맥 안에서만, 그렇지 않으면 해당 객체의 문맥 안에서만 호출할 수 있다. 함수는 메서드에 비해서 더 일반적이다. 함수는 특정한 클래스나 객체에 소속된 것이 아니다. 따라서 모든 인스턴스 메서드는 인수 중의 하나가 객체인 함수로 볼 수 있다.

자바는 메서드만 포함하고 있는 데, 메서드는 자바에서 1등 시민이 아니다. 메서드를 다른 메서드에 인수로 전달하거나, 메서드를 리턴받거나, 메서드를 다른 변수의 값으로 할당하거나 하는 일을 수행할 수 없다.

대신 익명 내부 클래스anonymous inner class가 실질적으로 이러한 함수의 기능을 보완해 주는 보자기wrapper 역할을 수행한다. 자바 메서드 중에는 내부에 한 개의 메서드만 정의하고 있는 인터페이스를 받아들이는 것이 많다. 다음은 AWT/Swing 애플리케이션에서 ActionListener를 정의하고 있는 예다.

```
package functions;
import java.awt.*;
import java.awt.event.*;

class HelloButtonApp2 {
```

```
        private final Button button = new Button();

    public HelloButtonApp2() {
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Hello There: event received: " + e);
            }
        });
    }
}
```

버튼이 어떤 작업을 수행하기 원하면, actionPerformed라는 메서드 한 개만 포함하고 있는 ActionListener 객체를 정의해야 한다. 이러한 메서드의 구현을 위해서, 익명 내부 클래스를 이용했다.

자바 API에서는 이렇게 단 한 개의 추상적인abstract 메서드를 정의하는 인터페이스를 이용하는 기법을 흔히 사용한다. 이러한 메서드는 특정한 사건이 발생했을 때, 실행되는 클라이언트 코드를 구현하는 데 사용하기 때문에 콜백callback 메서드라고도 한다.

자바 API의 세계에는 ActionListener처럼 특정한 목적을 위해서 작성되는 코드가 수백 가지나 있다. 이런 식의 목적을 위해서 존재하는 코드는 개발자들이 공부해야 하는 분량을 엄청나게 증가시킨다. 이런 코드를 익히려면 직접 자바 문서를 읽거나 적어도 IDE가 그러한 내용을 기억하고 있도록 만들어야 한다. 우리는 추상이라는 것이 뭔가 좋은 것이라고 알고 있다. 그렇지 않은가? 그렇다면 이처럼 특정한 목적을 위해서 작성되는 산만한 함수 객체들을 더욱 추상화하는 방법을 알아보도록 하자.

우선 A라는 타입의 인수를 받아들이고 void를 리턴하는 함수를 포함하는 다음과 같은 인터페이스가 있다고 하자.

```
package functions;

public interface Function1Void<A> {
   void apply(A a);
}
```

이 제네릭 메서드의 이름은 어떤 식으로 불러도 상관없지만, 나는 'apply'라고 부를 것이다. 함수를 호출하는 것을 대개 인수들에 대해서 함수를 적용한다고 말하는 함수형 프로그래밍의 관행을 고려했을 때, 이 상황에서도 역시 apply라는 이름이 어울리기 때문이다.

이제 자바에서 사용하는 추상 윈도우 툴킷AWT(Abstract Window Toolkit)의 함수적 버전인 java.fawt.Component라는 객체가 있다고 하고, 그것은 ActionListener 대신 다음과 같이 Function1Void의 객체를 인수로 받아들인다고 가정해보자.

```
package functions;
import java.fawt.*;
import java.fawt.event.*;

class FunctionalHelloButtonApp {
   private final Button button = new Button();
   public FunctionalHelloButtonApp() {
      button.addActionListener(new Function1Void<ActionEvent>() {  // 1
         public void apply(ActionEvent e) {                        // 2
            System.out.println("Hello There: event received: "+e);
         }
      });
   }
}
```

변경된 줄은 1과 2라고 표시된 설명문으로 강조했다. 그 부분을 제외하면 코드의 내용은 앞에서 보았던 것과 동일하다.

첫 번째 예제에서 본 ActionListener처럼, addActionListener 메서드에 미리 정의되어 있는 특정한 타입을 인수로 전달한 것은 부적절한 객체가 전달되는 것을 원천적으로 막아주는 장점이 있다고 생각하는 사람도 있을 것이다. 인터페이스와 메서드의 이름을 미리 특정한 이름으로 정해놓으면 코드를 읽는 사람을 위한 API 문서를 작성하는 데에도 도움이 된다고 생각하는 사람도 있을 것이다. 하지만 이러한 주장은 둘 다 사실과 거리가 멀다.

우선 추상에 특별한 이름을 미리 정해놓는 것은 사용자가 잘못된 내용을 구현하는 것을 막는 데 아무런 도움을 주지 않는다. 그리고 문서화에 대해서 말하자면, addActionListener 메서드는 이름을 미리 정하든 말든 상관없이 자신이 기대하는 바를 문서화해야 한다(뒤에 나오는 리스코프 치환 원칙에서 자세히 살펴보게 될 것이다). Function1Void〈ActionEvent〉에 전달되는 제네릭 타입 파라미터는 addActionListener 시그너처의 일부다. 그리고 그것은 사용자에게 꼭 필요한 정보를 잘 전달해주고 있다.

개발자들이 JDK의 모든 곳에서, 미리 특정한 방식으로 지어진 이름이 아니라 Function1Void〈A〉처럼 보편적인 인터페이스를 사용하는 데 익숙해진다면, 라이브러리의 곳곳에서 사용되는 일회성 인터페이스의 내용을 일일이 기억할 필요가 없다. 하나의 이름만 존재하기 때문이다. Function1Void〈A〉라는 단 한 개의 보편적인 이름은 다양한 함수를 포함할 수 있는 함수 보자기function wrapper다.

이제 우리는 재사용하기 쉬운 새로운 추상을 도입하였다. 이젠 addActionListener에 전달되는 특정한 타입의 이름을 별도로 기억할 필요가 없다. 모든 장소에서 Function1Void를 사용하면 된다. 심지어 메서드의 이름도 일일이 기억할 필요가 없다. 그것은 언제나 apply이기 때문이다.

이렇게 추상적인, 단 한 개의 함수를 광범위한 문맥에서 사용하면 개발자가 별도로 공부하고 기억해야 하는 내용이 엄청나게 줄어든다는 사실은 충격적이다. 이제 특정한 인터페이스의 이름을 기억할 필요가 없게 되었다. 함수의 이름이 아니라, 함수가 수행하는 일이 무엇인지만 알면 충분하기 때문이다.

2.1.3 람다와 클로저

앞에서 JDK에 포함되어 있는(특정한 인터페이스나 메서드의 이름과 관련된) 불필요한 복잡성이 줄어들도록 만들었다(혹은 그렇게 한 것처럼 시늉을 내었다). 하지만 new Funcion1Void〈ActionEvent〉 {...}처럼 코드의 내용을 길게 적어야 하기 때문에 문법 자체는 산만하다. 익명의 함수를 그냥 인수의 리스트와 함수의 본문만 가지고 만들 수 있다면 더 간단하지 않을까?

요즘에는 많은 프로그래밍 언어가 이러한 기능을 지원한다. 수년 동안의 지루한 논쟁 끝에, JDK 8은 람다lambda라고 불리는 익명 함수를 위한 문법을 제공하기로 했다([ProjectLambda]와 [Goetz2010]을 참조).

자바에 추가될 것으로 보이는 문법의 구조는 다음과 같다.

```
public FunctionalHelloButtonApp() {
  button.addActionListener(
    #{ ActionEvent e -> System.out.println("Hello There: event received: "+e) }
  );
}
```

#{...} 표현은 람다 표현식의 리터럴 문법literal syntax이다. 인수의 리스트는 화살표(->)의 왼쪽에 있고, 함수의 본문은 화살표의 오른쪽에 있다. 이러한 문법이 일상적으로 반복되는 상용적인 코드를 얼마나 많이 없앨 수 있는지 직접 확인해보라!

NOTE_ 람다라는 용어는 익명 함수의 다른 표현이다. 이 이름은 람다 계산법에서 함수를 나타내는 데 사용되는 그리스 문자 λ에서 유래되었다.

논의를 더 완벽히 하기 위해서, 함수 타입의 예를 하나 더 보도록 하자. 다음 예제가 보여주는 함수는 A1과 A2라는 타입을 사용하는 두 인수를 받아들이고 void가 아니라 타입 R 값을 리턴한다. 이 예는 익명 함수를 위해서 사용되는 스칼라 타입에서 따온 것이다.

```
package functions;
public interface Function2<A1, A2, R> {
   R apply(A1 a1, A2 a2);
}
```

불행하게도 이 함수는 다른 애리티arity를 갖는 모든 함수에 대해서 별도의 인터페이스를 정의해야 한다(여기에서 애리티는 인수의 수를 의미한다). 실제로는 R 값에 2를 곱해야 한다. void가 리턴되는 경우와 void가 아닌 구체적인 값이 리턴되는 경우를 모두 고려해야 하기 때문이다. 이러한 노력은 실제로 널리 사용되는 프로젝트에서 어느 정도 정당화되었다. [Functional Java] 프로젝트는 사용자들을 위해서 이러한 인터페이스를 이미 정의했다.

여/기/서/잠/깐 클로저

클로저는 함수 본문이 인수로 전달될 때, 혹은 함수가 자기 내부에서 정의된 것이 아니라 바깥에서 정의된 변수를 의미하는 자유변수free variables를 사용할 때 만들어진다. 코드를 실행하는 런타임은 그러한 자유변수를 "어떤 박스에 넣고 잠가서" 나중에 함수가 실제로 실행할 때, 사용할 수 있도록 만든다. 함수는 그러한 변수를 선언한 바깥 부분의 코드가 이미 실행되고 사라진 한참 후에 실행될지도 모른다. 자바는 내부 클래스를 통해서 클로저의 기능을 제한적으로 지원한다. 이러한 내부 클래스는 바깥 영역에 있는 변수가 final로 선언되었을 때에 한해서 사용할 수 있다.

2.1.4 고계함수

다른 함수를 인수로 받아들이거나 함수를 리턴하는 함수를 지칭하는 특별한 용어가 있다. 고계함수higher-order functions가 그것이다. 자바 메서드는 인수나 리턴하는 값으로 원시 값과 객체만 사용할 수 있다. 하지만 앞에서 정의한 Function 인터페이스를 이용해서 고계함수를 흉내 낼 수 있다.

고계함수는 추상을 만들고 특정한 행위를 구성할 때 매우 강력한 도구로 사용할 수 있다. 3장에서 고계함수가 List나 Map 같은 표준 라이브러리 타입에 대해서 거의 무제한적인 변화를 줄 수 있고, 재사용 가능성도 높인다는 사실을 확인하게 될 것이다. 사실 이 장의 시작 부분에서 언급했던 조합기는 고계함수다.

2.1.5 부수효과가 없는 함수

버그를 양산하는 복잡성의 원천 중에는 어떤 상태에 변화를 주는 함수, 즉 객체의 필드나 광역변수의 값을 변경하는 함수가 있다.

수학에서 함수는 어떤 부수효과도 낳지 않으며 부수효과에서 자유롭다. 예를 들어서, sin(x)가 어떤 일을 수행한다고 해도 그 결과는 전적으로 함수를 호출한 곳으로 리턴된다. 이 함수의 호출 때문에 외부의 상태가 변하는 일은 없다. 물론 실제 구현에서는 효율성을 높이기 위해서 앞에서 계산했던 값을 캐시에 저장하거나 하는 기법을 사용할 수도 있다. 이런 경우에는 캐시의 값이 변하는 부수효과가 발생하기도 한다. 하지만 함수를 호출하는 사람의 입장에서 보았을 때는 (스레드 안정성을 포함한) 외적인 부수효과가 없도록 만드는 것이 전적으로 함수를 구현하는 사람의 의도에 달려있다.

특정한 파라미터의 집합을 이용해서 함수를 호출하는 코드 자체를 그것이 리턴하는 값으로 대체할 수 있다면, 이를 참조투명성referential transparency이라고 부른다. 이는 부수효과를 갖지 않는 함수를 고려했을 때 매우 중요한 의미를 갖는다. 즉, 함수와 그것이 리턴하는 값을 계산이라는 측면에서 보았을 때 동의어에 해당하는 것이