Hanbit eBook

Realtime 94

코드의 재사용과 높은 수준의 테스팅을 원한다면

함수형 길들이기

Becoming Functional

조슈아 백필드 지음 / 이일웅 옮김

O'REILLY® 한빛미디어 Hanbit Media, Inc.

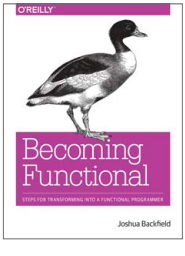
코드의 재사용과 높은 수준의 테스팅을 원한다면

함수형 길들이기

Becoming Functional

조슈아 백필드 지음 / 이일웅 옮김

O'REILLY® 한빛미디어 Hanbit Media, Inc.



이 도서는
Becoming Functional (O'REILLY) 의
번역서입니다

코드의 재사용과 높은 수준의 테스팅을 원하다면 **함수형 길들이기**

초판발행 2015년 3월 26일

지은이 조슈아 백필드 / **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 978-89-6848-743-9 15000 / **정가** 13,000원

총괄 배용석 / **책임편집** 김창수 / **기획·편집** 김상민
디자인 표지/내지 여동일, 조판 최송실
마케팅 박상용 / **영업** 김형진, 김진불, 조유미

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

지은이_ **조슈아 백필드**

조슈아 백필드Joshua Backfield는 MSSP 분야의 선도 기업인 델 시큐어웍스Dell SecureWorks, Inc.의 수석 소프트웨어 개발 엔지니어로, 다중 백엔드 프로세스와 다양한 내부 UI 도구의 설계, 개발을 담당하고 있습니다. 카본데일에 있는 서던 일리노이즈 대학교에서 전자 시스템 공학(학부)을, 드폴 대학교에서 컴퓨터 공학(석사)을 각각 전공했습니다. C, C++, 펄, 자바, 자바스크립트, 스칼라 같은 다양한 프로그래밍 언어를 구사하며, 지금도 더 많은 언어를 배우려고 노력 중입니다. C로 작성된 갖가지 애플리케이션을 스칼라로 전환하는 업무를 수행하며, 동료들에게 스칼라를 소개하고 함수형 프로그래밍의 개념을 가르치고 있습니다.

역자 소개

옮긴이_ **이일웅**

적잖은 세월 동안 대기업과 공공기관 등지를 전전하며 각종 프로젝트에 참여한 자바 웹 개발자다. 함수형 프로그래밍의 묘미에 푹 빠진 이후로는 스칼라, Akka, Play Framework에 심취하여 국내 저변 확대 및 지식 보급에 힘쓰고 있다.

- 개인 홈페이지: http://www.bullion.pe.kr

프로그래밍 세계에서 함수형functional 개념은 아주 오래된 역사를 갖고 있음에도 불구하고 상아탑에서 벗어나 현장에서 관심을 받기 시작한 지는 그리 오래되지 않았습니다. 그러나 해외의 많은 성공 사례가 증명하듯, 이제는 수천 개의 멀티 코어를 가진 클러스터링 환경에서 조금이라도 더 가볍고 빠른 멀티스레드 애플리케이션을 개발해야 하는 상황이므로 프로그래밍 스타일 역시 지난 수십 년을 지배했던 객체 지향 프로그래밍에서 진화할 시점이 아닌가 싶습니다.

그런데 함수형 프로그래밍이 기존의 프로그래밍 관념을 100% 뒤엎는 대변혁이라고 할 만한 것은 아니지만, 함수형 프로그래밍을 처음 접하는 개발자 입장에서는 손에 익어 버린 명령형 스타일의 코딩 습관이 머릿속에 워낙 단단히 박혀 있어 곧바로 사고방식을 전환하기가 어려운 게 사실입니다. 이 책의 독자 여러분 역시 저와 같은 자바 개발자인 경우가 대부분일 텐데, 똑같은 결과를 내는 코드임에도 상당히 색다른 방식으로 접근하고 함수를 매개로 한 구조로 작성하려는 시도가 처음에는 다소 거부감을 안겨 줄 수도 있습니다. 특히, 이른 시간에 많은 실적을 보여줘야 생존할 수 있는 국내의 척박한 개발자 업무 환경을 고려하면 한가한 사람들의 뜬구름 잡는 이야기처럼 들릴 수도 있습니다.

하지만 좀 더 시야를 넓게 가지고 함수형 스타일이 주목을 받게 된 이유가 무엇인지, 여러분 자신의 잊고 싶은 쓰라린 경험(예컨대, 멀티 쓰레드 애플리케이션의 동기화 오류로 인해 데이터가 뒤죽박죽된 원인을 찾다가, 아니면 시스템 오픈 직전 개발 시간의 몇 배를 투자하여 테스트/디버깅을 하느라 밤을 지새운 일들)과 잘 결부시켜 생각해 보면 함수형 프로그래밍의 독특한 매력과 가능성이 보이기 시작할 것입니다. 그리고 이 책의 책장을 넘기면서 그루비, 스칼라 같은 언어에 관심을 두고 학습을 계속하다 보면 폴리글랏polyglot 프로그래머로 한 걸음 발전한 자신을 발견하게 될 것입니다.

그러나 만사가 다 그렇듯이 함수형 개념이 완벽한 것은 아니며, 어떤 상황에서도 최적의 결과를 가져다주는 만병통치약은 아니므로 어디까지나 자신의 프로그래밍 세계를 더 확장하여 필요에 따라 기존 방식과 병용하여 문제 해결을 할 수 있는 지혜를 발휘해야 합니다. 스칼라의 창시자인 마틴 오더스키Martin Odersky 역시 자신의 저서에서 스칼라는 무조건 함수형 스타일을 강제하는 순수 함수형 언어도 아니고 직면한 문제에 따라 명령형과 함수형을 적절히 혼합하여 사용할 줄 알아야 한다고 말한 바 있습니다. 아무리 좋은 사상, 아무리 좋은 도구도 그것을 쓰는 사람의 역량과 지혜로움에 따라 그 결과는 크게 달라질 수밖에 없을 것입니다.

아무쪼록 본 역서가 더 많은 분들이 함수형 개념에 눈을 뜨고 관심을 갖게 되는 계기가 되었으면 하는 바람입니다. 재미있는 함수형 프로그래밍의 세계를 마음껏 즐기시기 바랍니다.

2015년 춘풍이 불어오기 시작하는 날에

이일웅

1. 국어 표준 맞춤법, 외래어 표기법 및 띄어쓰기를 준수한다.
2. 기술 용어, 제품명 등 고유 명사 형태의 원어는 최대한 한글로 음차하여 옮긴다(예: Scala → 스칼라, Groovy → 그루비 등). 그러나, 약자 형태로 된 원어나 그 밖에 한글 음차가 부적절한 경우 원어를 그대로 표기한다(예: PK 등).
2. 2번에서 한글 음차 시 최초 1회 영문을 위 첨자 형태로 병기하고, 이후 반복하여 등장할 경우 이를 생략한다. 그러나, 재등장 시 독자의 이해를 위해 필요하다면 다시 영문 병기를 한다.
3. 어디까지나 이 책을 구입하여 읽을 대상 독자들의 이해를 쉽게 하는 방향으로 번역한다. 따라서, 기술에 문외한인 일반인들에게 생소하게 들리는 말이라도 기술자들 간에는 많이 사용하여 익숙한 말이라면 억지로 풀어쓰지 않는다(예: 패턴 매치, 인스턴스, 인터페이스 등).
4. 이 책은 입문서의 성격임을 감안하여 '~ㅂ니다'체를 사용하되 극존칭 어법은 지양한다. 강사가 학생에게, 선배 기술자가 후배 기술자에게 최대한 친절하게 지식을 전수하는 형태로 기술함을 지향한다.
5. 예제 코드의 주석 및 기술적인 내용과는 무관한 상수 문자열은 한글 번역을 하되, 프로그램 로직을 파악하는 데 오히려 방해가 되거나 코드 가독성을 떨어뜨릴 수 있는 경우 원래 코드를 유지한다.

함수형 프로그래밍은 사실 최근에 등장한 개념은 아니지만, 최근에서야 프로그래머들 사이에서 폭넓은 관심을 받기 시작했습니다. 불변 변수immutable variable, 순수 함수pure function가 코드 디버깅에 유용하고, 고차 함수higher-order function를 쓰면 함수 내부 처리 로직을 추출하여 코딩량을 줄일 수 있다는 사실이 확인되었습니다. 표현적인expressive 코드는 이제 분명 대세인 것 같습니다.

이 책의 대상 독자

이 책은 함수형 프로그래밍에 관심이 있거나 기존 명령형 코드의 함수형 전환을 고려중인 분들을 위해 쓰였습니다. 명령형 또는 객체 지향 스타일의 코딩을 주로 해 온 프로그래머라면 부디 이 책을 읽고 앞으로는 함수형 코드를 작성했으면 하는 바람입니다.

이 책은 명령형 스타일의 패턴을 인지하고 함수형 스타일로 자연스럽게 전환하는 방법을 가르칩니다. 구체적으로 설명하기 위해 한빛증권이라는 가상 회사의 레거시 코드를 예로 들어 명령형에서 함수형 스타일로 코드를 어떻게 리팩토링하는지 살펴볼 것입니다. 이 과정에서 다음 언어를 사용합니다.

자바

자바 문법은 대부분 익숙하리라 생각합니다. 이 책에서는 1.7.0 버전을 씁니다.

그루비Groovy

기존 자바 문법을 최대한 유지하면서 완전한 함수형 언어로 전이하는 과정을 살펴볼 목적으로 사용합니다. 필요한 곳에서 중요한 문법은 따로 설명합니다. 2.0.4 버전을 씁니다.[01]

01 역자주_이 책을 번역하는 2015년 2월 현재, 최신 버전은 2.4.1입니다.

스칼라Scala

스칼라는 전환의 종착역이라 할 수 있는 완전한 함수형 언어입니다. 그루비와 마찬가지로 처음 등장할 때 문법을 설명하겠습니다. 사용 버전은 2.10.0입니다.[02]

자바 8은 안되나요?

어떤 분은 왜 자바 8은 안 쓰느냐고 하실지도 모르겠습니다. 하지만 이 책을 집필하는 시점에 자바의 안정 버전stable version은 7이고 가장 많이 사용하고 있습니다.[03] 저는 얼리 어댑터뿐만 아니라 가급적 많은 분이 이 책을 읽고 깨달음을 얻기 원하므로 자바 7로 시작하는 편이 현재로썬 접근성이 제일 낫다고 생각합니다.

이미 자바 8을 쓰시는 분들은 고차 함수 같은 개념을 그루비 언어로 바꾸지 않고 그대로 사용하면 됩니다.

수식 표기법 복습

함수형 프로그래밍은 수학과 밀접한 연관이 있으니 잠시 기본적인 수식 표기법을 복습하겠습니다.

수학에서 함수는 '함수명(파라미터[04]) = 몸체'의 형식으로 표현합니다. 수식 P-1은 아주 간단한 함수의 예입니다. 함수의 이름은 f, 파라미터는 x, 몸체는 x + 1이고, x + 1을 반환합니다.

---

02 역자주_현재 2.11.5 버전까지 출시되었습니다.

03 역자주_이미 자바 SE 8은 8u31 버전까지 업데이트되었고, 오라클은 2015년 4월 이후로 자바 7의 업데이트는 더는 제공하지 않는다고 밝혔습니다.

04 역자주_엄밀히 말해서, 파라미터(parameter, 매개변수)와 인자(argument, 인수)는 다른 개념입니다. 파라미터는 함수 선언(원형 정의) 시 사용하는 변수, 인자는 함수 호출 시 넘겨주는 값이나 변수를 가리키는데, 이 책의 원문에서는 이를 구분하지 않고 혼용하고 있습니다.

[수식 P-1] 간단한 수학 함수

$$f(x) = x + 1$$

if 문은 수학에선 밑으로 내려 나열합니다. 조건에 해당하는 연산을 목록으로 펼쳐놓는 것이지요. [수식 P-2]는 간단한 계산식 세트입니다. abs(x)는 x가 0보다 작으면 x * -1을, 그 외의 경우엔 x를 반환하는 함수입니다.

[수식 P-2] if 문에 해당하는 간단한 수식

$$abs(x) = \begin{cases} x * -1 & \text{if } x < 0 \\ x & \text{else} \end{cases}$$

총계는 시그마 기호로 나타냅니다. [수식 P-3]는 0부터(시그마 밑부분 n=0) x까지(시그마 윗부분 x) n이 계속될 때, 각각의 n을 더하라(시그마 우측 몸체로 정의)는 뜻입니다.

[수식 P-3] 간단한 총계 수식

$$f(x) = \sum_{n=0}^{x} n$$

명령형보다 함수형이 나은 이유

명령형, 함수형, 이벤트 주도형event-driven 등의 프로그래밍 패러다임은 저마다 장단점이 있고 호불호가 엇갈립니다. 아무래도 가장 일반적이고 많은 사람이 익숙한 방식은 명령형 프로그래밍입니다. 자바, C가 그렇게 설계된 대표적인 언어들입니다. 자바는 객체 지향 프로그래밍(OOP) 개념이 녹아 있긴 하지만, 명령형 패러다임의 느낌이 지배적인 언어입니다.

제가 소프트웨어 개발에 종사할 당시 "왜 제가 애써 함수형 프로그래밍을 배워야 하죠?"라고 질문하시는 분들을 많이 보았습니다. 제가 수행 중인 프로젝트 대부분이 스

칼라 같은 언어로 작성되었으니 그냥 "프로젝트가 함수형 언어 기반이니까요."라고 무심히 답할 수도 있었습니다. 하지만 한 발짝 물러나 정말 어떻게 대답하는 것이 정답일지 생각해봅시다.

지금까지 전 암호처럼 작성된 까닭에 정확히 무슨 기능을 하는지 이해하기 어려운 명령형 코드를 숱하게 봐 왔습니다. 명령형 스타일은 일반적으로 코드를 작성하고 원하는 대로 짜 맞추는 형태입니다. 구현 상세가 어떻든 다 알지 못해도 클래스 위에 클래스를 얹는 식으로 진행하죠. 그런데 이 과정이 계속되면 나중에 스파게티 면처럼 여기저기 남용된 클래스 코드가 넘치기 시작하면서 관리하기 어려운 거대한 코드 베이스가 됩니다.

그러나 함수형 프로그래밍으로 작성하면 코딩 전과 코딩 중에 스스로 구현할 내용을 잘 이해하게 됩니다. 그래서 어디를 추상화해야 할지 인지할 수 있고, 같은 기능의 코드임에도 몸집을 줄일 수 있습니다.

함수형을 OOP와 함께 쓰는 이유는?

OOP라고 하면 클래스의 개념을 가진 패러다임을 떠올리기 쉽습니다. 하지만 OOP로 작성하는 방법을 보면 실제로 OOP는 객체 안에 변수를 캡슐화하는 용도로 쓰입니다. 이런 코드는 사실 "위에서 아래로" 실행되는 명령형 코드입니다. 함수형 사고로 전환하면 다른 함수에 함수를 매개로 주고받는 경우를 자주 보게 됩니다.

함수형 프로그래밍이 OOP를 대체할 거라 생각하는 분들도 계시지만, 메서드[05]를 포

05 역자주_함수(function)와 메서드(method)는 분명히 다른 개념이지만, 이 책에서는 '어떤 로직을 처리하는 프로그램 코드'의 의미로 혼용하고 있습니다. 메서드는 객체 지향 관점에서 클래스에 부속된 멤버의 개념으로 생각하고, 함수형 프로그래밍의 관점에서는 모든 것이 함수라고 보면 거의 맞습니다.

함한 객체를 사용하는 한 OOP는 계속 쓰게 될 겁니다. 그러나 이 메서드는 좀 더 순수하고 테스트하기 좋은 함수를 얻을 수 있게 대개 정적인 메서드를 호출하게 됩니다. 따라서 OOP를 교체하는 것이라기보다는 함수형 구조 내에서 객체 지향 설계를 차용하는 것이라 할 수 있습니다.

함수형 프로그래밍은 왜 중요한가?

자바에서 디자인 패턴 같은 개념은 일상적인 프로그래밍에 있어서 필수적이기 때문에 그것을 빼놓고는 아무것도 논할 수가 없습니다. 그래서 함수형 스타일이 제법 오래된 개념임에도 메인 프로그래밍 패러다임의 배경 지식 역할을 해왔다는 사실이 역설적이지만 흥미롭습니다.

그런데 과연 그렇게 오랜 역사를 자랑하는 함수형 프로그래밍이 요즘 들어 중요하게 인식되는 이유는 무엇일까요? 웹 사이트 구축이 유행처럼 번졌던 닷컴 시절로 돌아가 일반적으로 애플리케이션이라 불렸던 것들을 떠올려 봅시다. 작동만 되면 아무도 구현 언어와 패러다임 따위는 신경도 쓰지 않았습니다.

지금은 요구 사항과 기대 수준이 복잡하고 난해하기 때문에 엔지니어들은 수학 함수를 세심하게 다룰 줄 알아야 강력한 알고리즘을 미리 설계할 수 있고 다른 개발자들은 제한된 시간 내에 알고리즘을 구현할 수 있습니다. 기본 수학에 충실한 알고리즘일수록 이해하기 쉽습니다. 함수형 프로그래밍을 하다 보면 함수에 수학을 적용하게 됩니다. 도함수, 극한, 적분 등의 개념은 함수의 오류를 진단하는 데 분명 큰 도움이 될 것입니다.

덩치가 큰 함수는 테스트하기도 어렵고 가독성도 떨어집니다. 소프트웨어 개발자라면 종종 수많은 기능 요구 사항을 함수 하나에 몰아넣는 식으로 코딩했던 경험이 있을 겁니다. 하지만 이렇게 거대하고 복잡한 함수일수록 내부 구현 로직을 추출해서 여러 개

의, 작고 이해가 쉬운 함수들로 빼내면 코드의 재사용성도 좋아지고 높은 수준의 테스트가 가능해집니다.

코드 재사용성과 높은 수준의 테스팅, 이 두 가지는 함수형 언어로의 전환이 가져다줄 가장 중요한 선물입니다. 하나의 함수로부터 전체 기능 덩어리를 추출할 수 있다면 나중에 지루한 카피-앤-페이스트 작업을 하지 않아도 기능 변경이 쉽습니다.

이 책의 표기법

팁, 제안, 기타 일반적인 내용은 여기에 적습니다.

경고나 유의 사항은 여기에 적습니다.

수학 경고

수학 이야기를 해야 할 때는 사전에 경고를 하겠습니다. 수학 공식과 정말 친하지 않은 분들은 "수학 표기법 복습"을 참고하세요.

예제 코드 내려받기

보조 자료(예제 코드, 연습 문제 등)는 https://github.com/jbackfield/BecomingFunctional에서 내려받을 수 있습니다.

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾기도 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
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 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 좀 더 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한, 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2 무료로 업데이트되는 전자책 전용 서비스입니다

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

3 독자의 편의를 위해 DRM-Free로 제공합니다

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

4 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다

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

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

chapter 1 소개 001

1.1 함수형 프로그래밍 개요 001
1.1.1 일급 함수 002
1.1.2 순수 함수 002
1.1.3 재귀 002
1.1.4 불변 변수 002
1.1.5 느슨한 계산 002
1.1.6 문장 003
1.1.7 패턴 매치 003
1.2 함수형 프로그래밍과 동시성 003
1.3 정리하기 004

chapter 2 일급 함수 007

2.1 한빛증권 입사를 환영합니다! 007
2.2 함수를 객체로 010
2.2.1 If-Else 문을 이용한 리팩토링 011
2.2.2 필드를 추출하기 위해 함수 객체를 이용 013
2.3 익명 함수 020
2.3.1 람다 함수 021
2.3.2 클로저 026
2.4 고차 함수 029
2.5 그루비를 이용한 get 함수 리팩토링 031
2.6 정리하기 033

chapter 3 순수 함수 035

3.1 입력에 따라 출력이 달라진다 035
3.2 함수를 순수하게 040
3.3 부수 효과 045
3.4 정리하기 050
3.4.1 그루비로 갈아타기 051

chapter 4 불변 변수 057

4.1 가변성 057
4.2 불변성 064
4.3 정리하기 071

chapter 5 재귀 073

5.1 재귀 개요 075
5.2 재귀 078
5.3 꼬리 재귀 081
5.4 countEnabledCustomersWithNoEnabledContacts 함수 리팩토링 082
5.5 정리하기 085
5.5.1 스칼라 입문 086

chapter 6 조급한 계산과 느긋한 계산 089

6.1 조급한 계산 090
6.2 느긋한 계산 091
6.3 느긋함이 문제가 될 수도 있습니다 097
6.4 정리하기 102

chapter 7 문장 105

7.1 완전한 함수형 프로그래밍 언어의 세계로! 106
7.2 단순문 106
7.3 블록문 109
7.4 모든 게 다 문장이다 112
7.5 정리하기 122

chapter 8 패턴 매치 125

8.1 단순 매치 125
8.2 단순 패턴 128
8.3 리스트 추출 131
8.4 객체 추출 134
8.5 패턴 매치로 전환 136
8.6 정리하기 139

chapter 9 함수형 OOP 141

9.1 정적 캡슐화 141
9.2 객체는 그릇이다 144
9.3 코드는 데이터다 146
9.4 정리하기 149

chapter 10 결론 153

10.1 명령형에서 함수형으로 153
10.1.1 고차 함수의 도입 154
10.1.2 기존 메소드를 순수 함수로 전환 154
10.1.3 루프문을 재귀/꼬리재귀로 전환 155
10.1.4 가변 변수를 불변 변수로 전환 155
10.1.5 다음 단계는? 155
10.2 새로운 디자인 패턴 156
10.2.1 동시성을 고려한 메시지 전달 156
10.2 2 옵션 패턴(널 객체 패턴의 확장판) 156
10.2.3 객체를 싱글톤 메소드의 순수성으로 대체 158
10.3 총정리 158
10.4 정리하기 168

부록

부록.1 자바 8/그루비/스칼라 설치 171
부록.2 자바 8 171
부록.3 그루비 173
부록.4 스칼라 174

chapter 1

소개

예제 코드를 소개하기 전에 먼저 함수형 프로그래밍이란 무엇인지 살펴봅니다. 구체적으로는 함수형 프로그래밍을 구성하는 요소들과 수학과의 연관 관계에 대해 자세히 다룹니다.

함수형 프로그래밍의 기원은 LISP 시절로 거슬러 올라가는데, 1977년 존 배커스John Backus가 "프로그래밍, 폰 노이만von Neumann 스타일에서 벗어날 수는 없는가?- 함수형 스타일과 프로그램 대수학"이란 제목의 논문으로 튜링상을 받기 전까진 사실상 패러다임의 명칭 자체도 형성되지 않았습니다. 배커스는 이 논문에서 대수 방정식 조합으로 애플리케이션을 제작하는 문제에 대해 여러 가지 관점에서 논하였습니다.

1.1 함수형 프로그래밍 개요

함수형 프로그래밍이 정확히 무엇인가 하는 문제는 여전히 논란의 대상이 되고 있지만, 많은 사람이 전반적으로 인정하는 특성이 있습니다.

- 일급 함수First-class functions
- 순수 함수Pure functions
- 재귀Recursion
- 불변 변수Immutable variables
- 느긋한 계산Nonstrict evaluation
- 문장Statements
- 패턴 매치Pattern matching

1.1.1 일급 함수

일급 함수는 다른 함수를 인자로 받거나 반환할 수 있습니다. 생성한 함수 자체를 반환하거나 다른 함수에 전달하는 기능은 코드 재사용성과 추상화 측면에서 매우 유용합니다.

1.1.2 순수 함수

부수 효과side effects가 없는 함수를 순수 함수pure function라고 합니다. 부수 효과는 함수 본연의 기능 이외의 행위를 말하며, `println` 같은 외부 함수, 또는 전역 변숫값의 변경이 해당합니다. 함수에 인자로 넘긴 변수를 그 함수 내부에서 바꾸는 것도 부수 효과입니다.

1.1.3 재귀

재귀는 알고리즘 코드를 더욱 짧고 간결하게 작성할 수 있게 해줍니다. 또 재귀를 이용하면 함수의 입력부만 신경 써서 코드를 실행할 수 있는데, 현 단계의 반복과 반복을 계속할지 여부, 두 가지 문제만 집중하면 됩니다.

1.1.4 불변 변수

불변 변수immutable variable는 한번 할당하면 다시는 변경할 수 없습니다. 불변성은 구현하기가 까다로워 보이지만, 애플리케이션 내부의 특정 시점에 한하여 상태가 바뀐다는 전제하에 방법을 알아보겠습니다.

1.1.5 느슨한 계산

느슨한 계산Nonstrict evaluation이란, 변수를 선언만 해두고 계산은 나중으로 미루는 것을 말합니다. 조급한 계산Strict evaluation은 변수 선언과 동시에 값을 할당하는 것으로 우리가 지금까지 많이 봐 왔던 코드입니다. '느슨한'이란 말이 일단 선언만 해둔 상태에서 처음 누군가가 참조할 때 비로소 할당(계산)된다는 걸 의미합니다.

1.1.6 문장

문장은 값을 반환하는, 계산 가능한evaluable 코드 조각입니다. 어떤 종류든 반환할 값을 가졌는지 잘 따져 보아야 합니다. 코드는 한 줄 한 줄 문장으로 작성해야 애플리케이션 내부에서 부수 효과를 최소화할 수 있습니다.

1.1.7 패턴 매치

수학책에는 없는 패턴 매치는 함수형 프로그래밍에서 특정 변수들에 의존하지 않도록 도와줍니다. 보통 코드를 짤 때 객체 안에 변수들을 여러 개 캡슐화시키죠. 패턴 매치를 이용하면 좀 더 효과적으로 객체에서 구성 요소를 추출하고 타입을 체크할 수 있습니다. 변수를 더 쓰지 않아도 문장을 단순하고 간결하게 만들어 줍니다.

1.2 함수형 프로그래밍과 동시성

동시성Concurrency은 병렬 단위로 작업을 처리하는 개념인데, 제대로 설명하자면 따로 책을 써야 할 정도로 방대하므로 이 책에서는 다루지 않습니다. 함수형 프로그래밍이 동시성 이슈를 전부 해결할 구세주처럼 이야기하는 분들이 있는데, 꼭 그렇지는 않습니다. 다만, 함수형 프로그래밍의 개념을 잘 이해하고 적용하면 동시성을 다루는 데 효과적인 잘 짜인 패턴을 만들 수 있습니다.

예를 들어 메시지 패싱message passing 같은 기법을 사용하면 어떤 쓰레드thread가 메시지 수신 전에 다른 쓰레드를 차단하지 않도록 함으로써 독립적인 쓰레드를 더 많이 생성할 수 있습니다.

또 불변성과 같은 특징은 전역 상태를 정의하고 부분적인 상태 변화 또는 쓰레드 간 주요 동기화가 아닌 전역적인 상태 변이가 가능하도록 해줍니다.

1.3 정리하기

이 장에서는 함수형 프로그래밍의 중요한 개념을 개괄적으로 살펴보았습니다. 지금은 "이런 개념들만 가지고 진짜 시작할 수 있을까?"라고 의문을 품는 독자분들도 계실 겁니다. 계속 읽다 보면 함수형 프로그래밍의 특성들을 여러분의 코드에 어떻게 접목할 수 있을지 아이디어가 떠오를 것입니다.

장마다 먼저 개념을 소개하고 한빛증권 사의 예제 코드를 리팩토링하는 과정에서 구체적인 그림을 그려봅니다. 이 책의 예제 코드엔 "드라이버 코드driver code"가 없습니다. 여러분 스스로 자바 `main` 함수를 만들어 간단히 테스트할 정도의 지식은 있다고 가정하겠습니다. 드라이버 코드를 일부러 뺀 이유는 두 가지입니다.

첫째, 여러분 손으로 직접 코드를 짜서 테스트해 봐야 합니다. 예제를 훑어보기만 해서는 개념을 잘 이해할 수 없고 바람직한 함수형 프로그래머가 되기 어렵습니다.

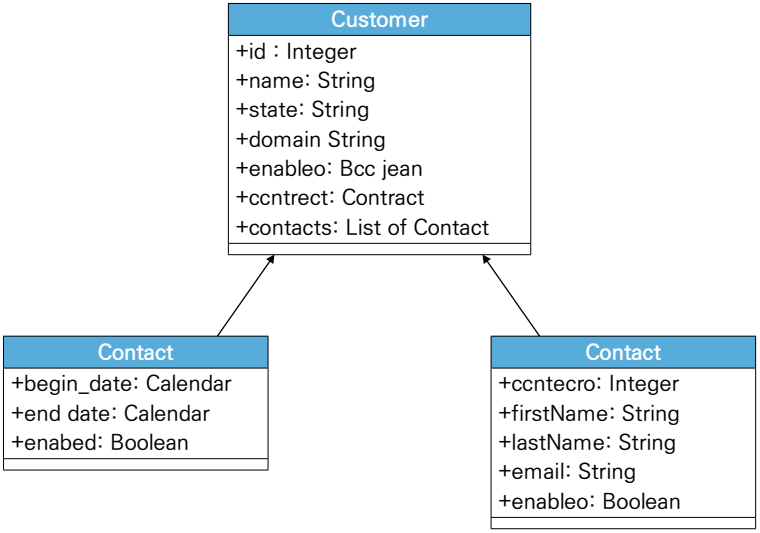
둘째, 여러분이 드라이버 코드에 집중하지 않았으면 합니다. 리팩토링한 코드를 단지 호출만 하는 이 책의 주제와는 별 상관없는 코드를 작성할 때, 종종 드라이버 코드의 리팩토링을 놓치거나 코드를 장황하게 만드는 경향이 있습니다. `Customer` 객체를 열 개, 스무 개 생성하는 드라이버 코드를 한 무더기 찍어내는 게 과연 큰 도움이 될까요?

각 언어로 작성된 예제 코드는 모두 컴파일 및 실행 가능하며 외부 패키지에 의존하지 않습니다. 몇몇 개념서나 언어 책을 보면 저자가 서드파티third-party 패키지를 내려받으라고 강요하는 듯한 느낌을 받는데, 저는 이런 태도가 정말 싫습니다. 이 책의 의도는 어쨌든 여러분이 중요한 프로그래밍 언어로 함수형 개념을 이해할 수 있게 가르치는 것입니다.

역자 NOTE 1

다음 장부터 여러분이 입사(?)하여 '한빛몰 시스템 함수형 전환 프로젝트'에 본격 참여하시기 전에, 데이터 구조 및 몇몇 업무 용어들에 대해 간단히 정리하겠습니다(이 부분을 건너뛰어도 책장을 넘기는 데 전혀 문제는 없지만, 예제 코드에 등장하는 각종 변수를 이해하는 데 조금은 도움이 되리라 생각합니다. 또한, 본 역서에서 이 변수들과 관련하여 어떻게 비즈니스적인 의미를 해석했는지 밝혀두어야 차후 독자 여러분들의 혼동이 없을 것 같습니다).

그림 1 간단히 일부 속성만 추출한 클래스 다이어그램



앞의 그림은 예제 코드에 빈번히 등장하는 주요 3개 엔티티(고객Customer, 계약Contract, 연락처Contact)를 간략히 클래스 다이어그램으로 나타낸 것입니다. 여기서 고객당 하나의 계약 정보가 존재하고, 다수의 연락처 정보를 가질 수 있는 구조라는 걸 알 수 있습니다. 실제 비즈니스와는 다소 거리가 있는 모델링이지만, 이 책의 주제가 데이터 모델링이 아니라는 점을 고려하면 이 정도만 이해해도 충분합니다.

다음 표는 엔티티별 필드 목록을 정리한 것입니다.

고객(Customer) 필드 목록

필드명	설명
id	식별자: 단순 일련번호입니다.
name	성명
state	국가: 우리나라에는 주(state)의 개념이 없으므로 국가로 번역하겠습니다.

필드명	설명
domain	도메인: 이메일 @ 기호 뒤편에 있는, 소속 기관의 인터넷 도메인을 말합니다.
enabled	이용 여부(true/false): 실제로 한빛몰에 가입하여 이용 중인 고객인지를 나타내는 것으로 합니다. true면 '이용 고객', false면 '휴면 고객'입니다.

계약(Contract) 필드 목록

필드명	설명
begin_date	시작 일자: 계약이 발효되기 시작한 날짜입니다.
end_date	종료 일자: 계약이 만료되는 날짜입니다.
enabled	유효 여부(true/false): 계약 상태가 현재 유효한지 나타냅니다. true면 '유효 계약', false면 '해지 계약'입니다.

연락처(Contact) 필드 목록

필드명	설명
contact_id	연락처 ID: 단순 일련번호입니다.
firstName	이름
lastName	성씨
email	이메일 주소
enabled	사용 여부(true/false): 사용 중인 연락처인지 나타냅니다. true면 '현재 연락처', false면 '옛날 연락처'입니다.

노파심에서 말씀드리지만, Contract와 Contact가 철자가 비슷하여 눈이 피로한 상태에서 예제 코드를 보고 있으면 헷갈리기에 십상입니다. 프로그래밍 공부도 좋지만 가끔 주기적으로 맑은 공기를 마시고 종아리 운동을 해줘야 한다는 사실, 잊지 마세요!

chapter 2

일급 함수

함수형 프로그래밍 도서는 대부분 불변 변수로 시작하지만, 이 책은 일급 함수를 먼저 다룹니다. 여러분이 2장을 읽고 바로 내일이라도 배운 개념들을 실무에 활용할 수 있었으면 합니다.

일급 함수는 스스로 객체로 취급되는 함수입니다. 다른 함수에 파라미터로 전달하고 반환받을 수 있고, 그냥 변수로 저장할 수도 있는 함수입니다. 가장 유용한 함수형 프로그래밍의 특징이지만, 효과적인 사용법을 익히기가 가장 만만찮은 주제이기도 합니다.

2.1 한빛증권 입사를 환영합니다!

한빛증권에 입사하신 여러분을 환영합니다. 여러분의 함수형 프로그래밍 스킬에 감명받은 팀장님은 기존 코드를 "함수형"으로 전환하고 싶어 합니다. 자바로 개발한 한빛증권 시스템을 앞으로 그루비나 스칼라 같은 새로운 언어로 바꿀 계획도 구상 중입니다. "기존 코드를 모두 폐기하고 맨땅에서 출발하고 싶은 마음"은 굴뚝같지만, 회사 사정상 그렇게는 곤란하다고 하네요.

좋습니다, 이제 시작해 봅시다. 여러분이 받은 첫 미션은 이용 고객의 주소 목록을 조회하는 함수의 신규 개발입니다. 팀장님은 이미 같은 유형의 기능이 구현된 Customer.java에 코딩하라고 합니다([예제 2-1]).

[예제 2-1] Customer.java

```
import java.util.ArrayList;
import java.util.List;

public class Customer {

 static public ArrayList<Customer> allCustomers = new ArrayList<Customer>();
 public Integer id = 0;
 public String name = "";
 public String address = "";
 public String state = "";
 public String primaryContact = "";
 public String domain = "";
 public Boolean enabled = true;

 public Customer() {}

 public static List<String> getEnabledCustomerNames() {
  ArrayList<String> outList = new ArrayList<String>();
   for(Customer customer : Customer.allCustomers) {
    if(customer.enabled) {
     outList.add(customer.name);
    }
   }
  return outList;
 }

 public static List<String> getEnabledCustomerStates() {
  ArrayList<String> outList = new ArrayList<String>();
  for(Customer customer : Customer.allCustomers) {
   if(customer.enabled) {
    outList.add(customer.state);
   }
  }
  return outList;
 }

 public static List<String> getEnabledCustomerPrimaryContacts() {
  ArrayList<String> outList = new ArrayList<String>();
  for(Customer customer : Customer.allCustomers) {
   if(customer.enabled) {
    outList.add(customer.primaryContact);
   }
  }
```

```
  return outList;
 }

 public static List<String> getEnabledCustomerDomains() {
  ArrayList<String> outList = new ArrayList<String>();
  for(Customer customer : Customer.allCustomers) {
   if(customer.enabled) {
    outList.add(customer.domain);
   }
  }
  return outList;
 }

 /* TODO: main 함수는 알아서 추가하자! */
}
```

현행 코드를 보니 함수별로 네 가지 유형의 코드가 줄곧 반복되고 있군요.

- ArrayList 생성
- for 루프
- if 문
- return 문

함수 하나당 6개, 총 18개 라인의 코드가 중복입니다. 12개 라인을 그냥 카피 앤 페이스트copy-and-paste한 셈이죠.

DRY 원리란?

코드 중복을 피해야 한다는, DRY(스스로 반복하지 말라Don't Repeat Yourself) 원리는 이미 오래전에 등장한 개념입니다. 코드가 중복되면 유지 관리가 점점 어려워진다는 얘긴데 왜 그럴까요?

어떤 함수가 여러 번 중복해서 코딩되면, 차후 그 함수에 버그가 발견되어 수정할라치면 카피 앤 페이스트한 다른 함수들도 모조리 뒤져봐야 하기 때문입니다.

enabled 변수명을 바꾼다든지, 아니면 다른 필드로 대체하려고 폐기deprecate하려면, 4개 함수 모두 코드 수정은 불가피합니다. 그 와중에 getDisabled* 함수 4개 역시 추가로 만들어 달라고 한다면? 8개의 함수를 카피 앤 페이스트 해야 합니다.

이즈음 여러분은 약간 골치가 아파져 오면서 "내가 대체 지금 무슨 짓을 하고 있는 거지?" 생각하기 시작합니다. 그러고는 심호흡을 크게 한 번 하고 여러분 자신이 다름 아닌 함수형 프로그래머라는 사실과 카피 앤 페이스트 근절이 자신의 임무임을 다시 한 번 상기합니다. 자, 첫 단추는 함수를 객체로 바라보는 겁니다.

2.2 함수를 객체로

좀 전에도 언급했지만, 일급 함수는 다른 함수에 인자로 전달하고, 다른 함수로부터 반환받을 수 있습니다. 함수란 정확히 무엇일까요? 아주 일반적으로 말하면, 나중에 다시 쉽게 참조할 목적으로 캡슐화시킨 처리 작업입니다. 즉, '매크로macro' 같은 겁니다.

함수는 어떤 요소들로 구성될까요? 함수는, 식별하기 위한 '함수명name', 처리 대상 객체들이 포함된 '파라미터 리스트parameter list', 그리고 파라미터로 어떤 작업을 수행 후 결과를 반환하는 '몸체body'로 이루어집니다.

Customer.java의 `getEnabledCustomerNames` 함수를 볼까요?([예제 2-2]) 함수명은 `getEnabledCustomerNames`, 파라미터 리스트는 비어 있습니다. 몸체에는 `Customer.allCustomers` 리스트를 반복하면서 이용 고객에 한하여 성명 필드(`customer.name`)를 output에 추가하는 코드가 들어 있습니다. 마지막 라인에서 결과 리스트인 `outList`를 반환하네요.

[예제 2-2] Customer.getEnabledCustomerNames

```
public static List<String> getEnabledCustomerNames() {
  ArrayList<String> outList = new ArrayList<String>();
   for(Customer customer : Customer.allCustomers) {
    if(customer.enabled) {
     outList.add(customer.name);
    }
   }
  return outList;
```

```
}
```

2.2.1 If-Else 문을 이용한 리팩토링

[예제 2-2]와 같은 기능(outList에 필드를 추가하는 기능은 제외)의 새로운 함수, getEnabledCustomerField를 작성합니다. 일단 //코드넣을자리 정도로 주석 표시하고, 나중에 customer.name 필드를 추출해서 outList에 넣는 기능은 따로 넣겠습니다.

먼저 ArrayList 객체를 생성합니다.

```
public static List<String> getEnabledCustomerNames() {
 ArrayList<String> outList = new ArrayList<String>();
```

for 루프에서 if 문으로 이용 고객 여부를 체크합니다.

```
for(Customer customer : Customer.allCustomers) {
 if(customer.enabled) {
```

방금 전 언급했듯이, 리스트에 필드를 담는 코드는 //코드넣을자리에 붙힐 것입니다. 이제 if 문, for 루프문 블록을 닫고 outList를 반환합니다.

```
    //코드넣을자리
   }
  }
 return outList;
}
```

하나로 합쳐보면 [예제 2-3]과 같은 모습이 되겠군요.

[예제 2-3] //코드넣을자리 표시한 getEnabledCustomerField

```
public static List<String> getEnabledCustomerField() {
 ArrayList<String> outList = new ArrayList<String>();
 for(Customer customer : Customer.allCustomers) {
  if(customer.enabled) {
   //코드넣을자리
  }
 }
 return outList;
}
```

Customer에서 해당하는 필드가 무엇인지는 이미 알고 있으니 이 중 하나를 새로운 파라미터로 받고, 이어지는 if 문 어느 한 곳에 걸려서 결과 리스트에 추가되도록 짜면 됩니다.

[예제 2-4] if 구조의 getEnabledCustomerField

```
public static List<String> getEnabledCustomerField(String field) {
 ArrayList<String> outList = new ArrayList<String>();
 for(Customer customer : Customer.allCustomers) {
  if(customer.enabled) {
   if(field == "name") {
    outList.add(customer.name);
   } else if(field == "state") {
    outList.add(customer.state);
   } else if(field == "primaryContact") {
    outList.add(customer.primaryContact);
   } else if(field == "domain") {
    outList.add(customer.domain);
   } else if(field == "address") {
    outList.add(customer.address);
   } else {
    throw new IllegalArgumentException("잘못된 필드입니다");
   }
  }
 }
 return outList;
}
```

예외에 의한 타입 안전 보장

[예제 2-4]는 타입 체크 실패 시 `IllegalArgumentException` 예외를 던집니다. 인자가 미리 정해진 필드가 아니면 명시적으로 오류를 일으키는 것이지요.

그러나 좋지 않은 발상입니다. 필드 접근자accessors를 단순 문자열로 비교하면서 타입 안전을 회피하고 있기 때문입니다. `if` 문의 조건식뿐 아니라, 호출하는 메소드 입장에서도 철자가 조금이라도 틀리면 안 되는 불안한 구조입니다.

타입 안전을 보장하는 다른 좋은 방법들이 있습니다. 일례로, 유효한 값들이 나열된 열거enumeration를 정의해서 `if/else` 문에서 해당 값을 매치시키는 방법이 있습니다.

이렇게 해서 한 함수에 반복적인 기능을 통합시켰습니다. 나중에 추출할 필드가 추가되면 어떻게 할까요? 네, 필드를 체크하는 `if/else` 문을 추가하면 됩니다. 결국, 코드는 `if` 문으로 어지럽게 뒤덮이고 관리하기 어려워지겠죠. 객체에서 필요한 필드만 뽑아내는 간단한 함수가 있으면 참 좋겠습니다.

2.2.2 필드를 추출하기 위해 함수 객체를 이용

자바 인터페이스로 다른 함수에 전달할 함수를 추상화시킬 겁니다. 자바 8 버전의 제안서에도 기술되어 있듯이 여타 프로그래밍 언어는 함수를 객체로 다룹니다. 하지만 이 글을 쓰고 있는 현재 자바 7이 가장 안정된 배포 버전이므로, 다른 함수에 함수를 전달하는 기능은 인터페이스로 구현할 수밖에 없습니다.

여러분은 쓰레드로 실행할 함수를 캡슐화시킨 `Runnable` 인터페이스를 잘 알고 계실 겁니다. 이런 인터페이스가 여기서도 필요한데, 차이가 있다면 (필드를 추출할 대상인) 객체를 인자로 받아 객체(필드값)를 반환한다는 점입니다.

수학 경고

함수 a를 참조하여 어떤 계산을 수행 후 값을 반환하는 함수 f가 있습니다.

$$(x) = x^2 / a(x)$$

자, 이제 a가 아닌 b를 호출하도록 함수 f를 수정하고 싶습니다. 그렇다고 f'처럼 이름만 다른 함수를 또 만드는 건 중복이지요. 람다 계산식에서는 함수를 함수에 넘길 수 있습니다. a를 호출하지 말고 아예 직접 전해줄 수 있으면 더 좋겠지요? f를 다시 정의하겠습니다.

$$f(x, c) = x^2 / c(x)$$

이제는 a든 b든 f를 호출하기가 아주 간편해졌습니다. 값을 넣어서 한번 호출해볼까요?

$$f(20, a) = 20^2 / a(20)$$

함수는 무엇을 인자로 받아야 할까요? 호출 지점을 잘 따져보면 엄청나게 많은 if 문들을 아낄 수 있습니다. 이제부터 작성할 함수의 목표는 Customer 데이터를 String으로 전환, 즉 Customer 객체를 '받아' String을 '반환'하는 일입니다. 인터페이스부터 정의하겠습니다.

이름은 ConversionFunction이라 붙이죠.

```
private interface ConversionFunction {
```

다음은 '함수'의 진입점이 될 메소드를 정의할 차례입니다. 이 함수는 Customer를 인자로 받아 String을 돌려줍니다.

```
  public String call(Customer customer);
}
```

[예제 2-5]는 완성된 ConversionFunction 인터페이스입니다.

[예제 2-5] ConversionFunction 인터페이스

```
private interface ConversionFunction {
 public String call(Customer customer);
}
```

이 인터페이스는 나중에 좀 더 일반적인 형태로 다듬고 나서 별도 파일로 분리한 뒤 접근 제한자를 public으로 바꿀 겁니다. 일단 지금은 ConversionFunction 인터페이스로 거대한 if 문 덩어리를 제거하는 문제에만 집중합시다.

먼저 인자를 ConversionFunction 객체로 바꾸고, 그다음 반복적인 if/else 문들을 func.call(customer) 호출로 교체합니다. ConversionFunction의 call 메소드가 내부적으로 변환을 수행한다는 걸 기억하기 바랍니다. 이렇게 호출 및 결과를 담는 코드만 남게 되었습니다([예제 2-6]).

[예제 2-6] ConversionFunction를 인자로 받는 getEnabledCustomerField

```
public static List<String> getEnabledCustomerField(ConversionFunction func) {
 ArrayList<String> outList = new ArrayList<String>();
 for(Customer customer : Customer.allCustomers) {
  if(customer.enabled) {
   outList.add(func.call(customer));
  }
 }
 return outList;
}
```

이제 점점 머릿속이 함수형으로 바뀌려고 꿈틀대고 있을 겁니다. ConversionFunctions의 구현체는 어떤 모습일까요? 그냥 추출할 필드를 반환하는 기능이 전부겠죠. 이를테면, 다음 CustomerAddress 클래스는 Customer 객체를 받아 address 필드를 반환합니다.

```
static private class CustomerAddress implements ConversionFunction {
 public String call(Customer customer) { return customer.address; }
}
```

이제 getEnabledCustomerAddresses 함수를 만들어 팀장님을 기쁘게 해드릴 시간입니다. 내부에서 getEnabledCustomerField라는 새로운 메소드를 호출할 때 전환 역할을 담당할 함수를 인자로 넘깁니다. 혹여 나중에 enabled의 의미가 달라지더라도 코드는 한 곳만 손을 대면 되는 구조입니다.

```
public static List<String> getEnabledCustomerAddresses() {
 return Customer.getEnabledCustomerField(new CustomerAddress());
}
```

지금 당장은 필요 없지만, 만약 이용 고객 전체 리스트를 뽑아달라는 요건이 접수되면 어떡해야 할까요? 우리가 작성한 인터페이스는 Customer 객체를 받아 String을 반환하도록 고정된 형태이니 별로 도움이 안 될 것 같습니다. 따라서 제네릭 타이핑generic typing 기법으로 좀 더 추상화된 인터페이스가 필요합니다. [예제 2-7]처럼 이름을 Function1로 바꾸고 두 타입 파라미터(파라미터 A1와 반환 타입 B)를 받도록 인터페이스를 고쳐봅시다.

[예제 2-7] 인자 1개를 받는 함수를 캡슐화한 인터페이스

```
public interface Function1<A1,B> {
 public B call(A1 in1);
}
```

타입 파라미터 명명 관습

왜 하필 이름을 Function1이라고 지었을까요? 다른 함수를 덮어씌우기wrapping 때문에 Function이라 명명한 것입니다. 여기서 숫자 1은 이 함수가 받는 인자의 개수를 나타냅니다. 인자가 2개인데 무슨 소리냐고요? 잘 보시면 두 번째 인자는 반환 타입입니다.

인자를 2개 취하는 함수([예제 2-8])도, 4개를 갖는 함수([예제 2-9])도 형태는 마찬가지입니다.

[예제 2-8] 인자 2개를 받는 함수를 캡슐화한 인터페이스

```
public interface Function2<A1,A2,B> {
 public B call(A1, in1,A2 in2);
}
```

[예제 2-9] 인자 4개를 받는 함수를 캡슐화한 인터페이스

```
public interface Function4<A1,A2,A3,A4,B> {
 public B call(A1 in1,A2 in2,A3 in3,A4 in4);
}
```

CustomerAddress는 이제 Function1<Customer,String> 인터페이스를 상속받습니다.

```
 static private class CustomerAddress implements Function1<Customer, String> {
  public String call(Customer customer) { return customer.address; }
 }
```

그리고 getEnabledCustomerField의 인자는 Function1로 고칩니다. Function1의 첫 번째 파라미터는 Customer로 고정이지만 두 번째 파라미터는 달라질 수 있으므로 B라고 해 둡시다. 결국, getEnabledCustomerField는 B를 인자로 받아 B 타입의 리스트를 반환하도록 파라미터화parameterize시킨 메소드가 되었습니다([예제 2-10]).

[예제 2-10] 제네릭 타입 함수 Function1을 받는 getEnabledCustomerField

```
public static <B> List<B> getEnabledCustomerField(Function1<Customer,B> func) {
 ArrayList<B> outList = new ArrayList<B>();
 for(Customer customer : Customer.allCustomers) {
  if(customer.enabled) {
   outList.add(func.call(customer));
  }
 }
 return outList;
}
```

이제 팀장님의 요구 사항을 다 충족했으니 다른 getEnabledCustomer* 함수들도 한데 모아볼까요? Function1 인터페이스를 상속한 클래스를 새로 만들고 getEnabledCustomer* 메소드가 적절한 클래스의 인스턴스로 Customer.getEnabledCustomerField( ) 메소드를 호출하도록 수정합니다. 이런 식으로 파일 나머지 부분도 죽 리팩토링한 뒤 [예제 2-11] 코드를 보면서 잘 작동하는지 살펴봅시다.

[예제 2-11] 초벌 리팩토링을 마친 Customer.java

```
import java.util.ArrayList;
import java.util.List;

public class Customer {

 static public ArrayList<Customer> allCustomers = new ArrayList<Customer>();
 public Integer id = 0;
 public String name = "";
 public String address = "";
 public String state = "";
 public String primaryContact = "";
 public String domain = "";
 public Boolean enabled = true;

 public Customer() {}

 private interface Function1<A1,B> {
  public B call(A1 in1);
 }

 static private class CustomerAddress implements Function1<Customer, String> {
  public String call(Customer customer) { return customer.address; }
 }

 static private class CustomerName implements Function1<Customer, String> {
  public String call(Customer customer) { return customer.name; }
 }

 static private class CustomerState implements Function1<Customer, String> {
  public String call(Customer customer) { return customer.state; }
```

```
 }

 static private class CustomerPrimaryContact implements Function1<Customer,
String>
 {
  public String call(Customer customer) { return customer.primaryContact; }
 }

 static private class CustomerDomain implements Function1<Customer, String> {
  public String call(Customer customer) { return customer.domain; }
 }

 static private class CustomerAsCustomer implements Function1<Customer,
Customer> {
  public String call(Customer customer) { return customer; }
 }

 public static List<String> getEnabledCustomerAddresses() {
  return Customer.getEnabledCustomerField(new CustomerAddress());
 }

 public static List<String> getEnabledCustomerNames() {
  return Customer.getEnabledCustomerField(new CustomerName());
 }

 public static List<String> getEnabledCustomerStates() {
  return Customer.getEnabledCustomerField(new CustomerState());
 }

 public static List<String> getEnabledCustomerPrimaryContacts() {
  return Customer.getEnabledCustomerField(new CustomerPrimaryContact());
 }

 public static List<String> getEnabledCustomerDomains() {
  return Customer.getEnabledCustomerField(new CustomerDomain());
 }

 public static <B> List<B> getEnabledCustomerField(Function1<Customer,B> func)
{
  ArrayList<B> outList = new ArrayList<B>();
  for(Customer customer : Customer.allCustomers) {
```

```
    if(customer.enabled) {
     outList.add(func.call(customer));
    }
   }
   return outList;
  }
}
```

아까 던졌던 질문으로 다시 돌아가죠. "전체 이용 고객 리스트가 필요하다면 어떻게 해야 할까?" 정답은 Customer 객체를 받아 Customer 객체를 반환하는 클래스를 추가하면 된다입니다([예제 2-12]).

[예제 2-12] Customer 주고 Customer 받기

```
static private class CustomerAsCustomer implements Function1<Customer,
Customer> {
 public String call(Customer customer) { return customer; }
}
```

Customer.getEnabledCustomerField(new CustomerAsCustomer( ))를 실행하면 전체 이용 고객 리스트가 멋지게 표시될 겁니다. 그런데 이렇게 클래스마다 일일이 이름을 붙이고 싶지 않고 완전한 클래스 형태로 정의할 필요가 없다면 어떻게 할까요? 익명 함수가 등장할 타이밍이네요.

2.3 익명 함수

익명 함수Anonymous function에는 람다 함수lambda function와 클로저closure, 두 가지 유형이 있습니다. 곧 설명해 드리겠지만, 클로저와 람다 함수는 서로 비슷하면서도 아주 미묘한 차이점이 있습니다. 함수는 함수명, 파라미터 리스트, 몸체, 반환까지 4개 파트로 구성된다고 했습니다. 그런데 굳이 함수명이 필요하지 않은 경우도 있기 때문에 스코프scope가 한정된, 잠깐만 존재하는 익명 함수가 고안되었습니다.