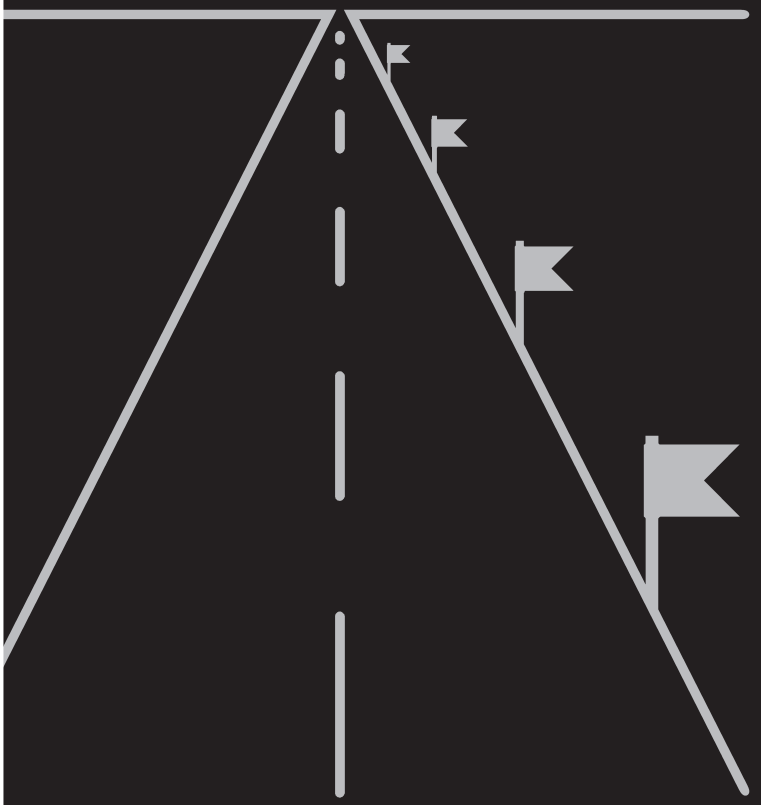
코틀린으로 배우는 함수형 프로그래밍

코틀린으로 배우는 함수형 프로그래밍

전자책 1쇄 발행 2023년 3월 24일 (종이책 초판 2쇄 반영) **지은이** 조재용, 우명인 **펴낸이** 한기성 **펴낸곳** (주)도서출판인사이트
편집 문선미 **등록번호** 제2002-000049호 **등록일자** 2002년 2월 19일 **주소** 서울특별시 마포구 연남로5길 19-5 **전화** 02-322-5143 **팩스** 02-3143-5579 **블로그** https://blog.insightbook.co.kr **이메일** insight@insightbook.co.kr **ISBN** 978-89-6626-397-4

프로그래밍 인사이트



코틀린으로 배우는 함수형 프로그래밍

조재용 · 우명인 지음

인사이트

차례

지은이의 글

필자들은 함수형 프로그래밍을 주제로 스터디 그룹을 운영하고, 많은 세미나를 진행해 왔습니다. 스터디와 세미나를 진행하면 늘 듣게 되는 질문이 있는데, 그중 몇 가지를 추려 답을 해 보았습니다. 그리고 이 책에 관심이 있는 사람이라면 누구나 궁금해할 만한 내용을 질문 형태로 추가했습니다.

Q. 최근 함수형 프로그래밍에 대한 관심이 커지고 있습니다. 그 배경은 어디에 있나요?

A. 함수형 프로그래밍에 대한 관심이 최근 들어서 높아지기는 했지만, 사실 함수형 프로그래밍은 상당히 오래된 기술입니다. 1930년대에 이론적 기반이 만들어졌고, 1980년대에 함수형 언어가 탄생했습니다. 그런데 최근 몇 년 사이에 함수형 프로그래밍은 개발자가 갖춰야 할 기본 스킬이 되었습니다.

함수형 프로그래밍이 등장하게 된 근본적인 원인은 갈수록 크고 복잡해지는 프로그램에 있습니다. 프로그램이 복잡해질수록 동시성 프로그래밍과 비동기 프로그래밍이 요구되는데, 동시성 프로그래밍과 비동기 프로그래밍은 상태를 가진 객체를 관리하기 어렵게 하고, 유지보수도 어렵게 합니다. 그에 대한 대처 방안이 연구되고 있을 때, 스칼라라는 함수형 언어가 등장합니다.

스칼라가 나온 2004년에는 하스켈이라는 함수형 언어가 이미 있었습니다. 하지만 언어에 대한 접근성이 매우 낮고 기존에 자바로 만들어둔 시스템을 모두 버려야 하는 현실적인 문제로 대중화되지 못한 상태였지요. 스칼라는 자바와의 호환성을 지원했고 객체지향 프로그래밍까지 가능한 하이브리드형 언어이기 때문에, 실제 프로젝트에 사용될 수 있었습니다. 스칼라의 등장으로 함수형 프로그래밍이 대중화될 수 있었던 것이지요.

하지만 아쉽게도 대중화의 영향이 한국에 미치지는 못했습니다. 당시만 하더라도 함수형 언어나 스칼라에 대한 한국어 콘텐츠가 많지 않았고, 대기업을 중심으로 형성된 한국의 개발 문화에서는 위험을 감수하고 자바로 만들어진 생태계를 바꾸려는 요구가 없었기 때문입니다. 한국에서는 자바8에 함수형 기능들이 들어오면서

부터야 본격적인 함수형 프로그래밍의 대중화가 시작되었습니다. 자바8이 출시된 지도 벌써 5년이 지났습니다. 함수형 프로그래밍은 이제 더 이상 생소한 개념이 아닙니다. 우리가 사용하는 대부분의 라이브러리와 플랫폼이 함수형 프로그래밍을 사용해서 만들어졌고, 함수형 프로그래밍을 지원하고 있습니다.

Q. 함수형 프로그래밍을 꼭 해야 하나요?

A. 네, 함수형 프로그래밍을 해야 하는 이유를 세 가지 키워드로 설명해 보겠습니다. 그 키워드는 불변성, 간결함, 안정성입니다(이 세 가지 장점과 영향에 대해서는 본 책의 1장에서 자세하게 다루고 있습니다). 함수형 프로그래밍에서는 한번 생성한 객체는 누구도 바꿀 수 없습니다. 따라서 개발자의 실수가 줄어들고 안전한 프로그램을 만들 수 있습니다. 또한 불필요한 예외처리나 분기문이 줄어들고, 미리 선언된 함수들을 재사용하므로 코드 양이 크게 줄어듭니다. 코드가 적으니 유지보수 또한 용이해집니다.

프로그램은 점점 커지고 요구사항에 대한 변화가 잦아지면서 빠르고 효율적인 개발이 요구되고 있습니다. 이러한 요구들을 만족시키기 위해서는 코드의 간결함과 유지보수의 효율성을 가진 함수형 언어가 필요합니다. 대부분의 모던 언어나 오픈소스들은 기본적으로 함수형 프로그래밍을 지원합니다. 함수형 프로그래밍은 객체지향 프로그래밍이 그랬던 것처럼 이미 하나의 패러다임으로서 우리 개발 문화와 코드 속으로 들어왔습니다.

Q. 함수형 프로그래밍을 설명하기 위해서 코틀린을 선택한 이유는 무엇인가요?

A. 코틀린은 스칼라와 같은 하이브리드형 언어입니다. 함수적 특징과 함께 객체지향 프로그래밍도 가능하도록 설계되어 있습니다. 덕분에 하나의 언어로 명령형·객체지향형·함수형 예제를 모두 설명할 수 있습니다.

스칼라의 경우 문법에 많은 기능이 포함되어 자유가 높은 반면, 언어 자체를 이해하는 데 시간이 많이 걸립니다. 그래서 본래의 목적인 함수형 프로그래밍을 공부하기 전에 스칼라 언어를 공부하느라 힘을 빼는 상황이 생길 수 있습니다. 하지만 코틀린은 형태가 자바와 유사해 공부하기도 쉽고 문법도 간단합니다. 따라서 함수형 프로그래밍 자체에 집중하기 훨씬 수월합니다.

그리고 실전에서 프로젝트를 개발할 때 순수 함수형 언어를 쓰는 경우가 드뭅니다.

이 책에서는 순수 함수형이 아닌 언어로 함수형 프로그래밍을 할 때의 시행착오와 대처법도 다루기 때문에 실질적으로 많은 도움이 될 것입니다.

Q. 코틀린을 배운 적이 없는데 이 책을 봐도 될까요?

A. 예제를 모두 코틀린으로 구현하였지만, 이 책의 주제는 기본적으로 함수형 프로그래밍입니다. 객체지향 프로그래밍 경험이 있다면 코틀린을 모르더라도 잘 따라올 수 있을 겁니다. 코드를 이해하는 데 필요한 정도의 코틀린 문법은 2장에서 다룹니다.

Q. 이 책은 어떤 독자들에게 무슨 도움을 줄 수 있을까요?

A. 이 책은 특히 명령형·객체지향형 프로그래밍에 익숙해서 함수형 프로그래밍에 적응하기 어려운 프로그래머를 염두에 두고 작성했습니다. 이 책을 통해 함수형 프로그래밍에 대한 기본 개념을 이해하고, 많은 연습문제를 직접 풀어 보고 답을 확인할 수 있습니다. 함수형 프로그래밍을 체득하고 나면 프로그램을 작성할 때 함수형 사고를 통한 함수적 설계를 할 수 있습니다.

또한 함수형 라이브러리를 개발하거나 함수형 언어로 만들어진 오픈소스를 분석해야 하는 독자에게도 도움이 됩니다. 코틀린은 실용적인 기능을 많이 제공합니다. 이 책에서는 그 기능을 이용해서 함수형 프로그래밍을 구성하는 함수적 구조물들을 모두 직접 만들어 봅니다. 함수형 프로그래밍을 위해서 필요한 것을 모두 직접 만들어 보면서 함수형 프로그래밍을 체득할 수 있을 것입니다. 뿐만 아니라 함수형 라이브러리를 직접 설계하고 만들어 제공할 수 있는 기반도 다질 수 있습니다.

Q. 이 책을 효율적으로 공부하는 방법이 있다면 알려 주세요.

A. 책의 뒷부분에서 다루는 내용과 예제를 이해하려면 책의 앞부분에서 다루는 기본 개념을 알아야 합니다. 따라서 가능하면 이 책을 순서대로 공부하길 바랍니다.

개념을 설명할 때는 예제와 연습문제를 많이 활용했습니다. 단순히 언어가 제공하는 함수적 기능을 써보는 것을 넘어서 고차 함수, 함수적 자료구조, 타입 등을 직접 만들어 볼 수 있도록 작성했습니다. 따라서 함수형 프로그래밍 경험이 있더라도 반드시 본문 예제를 직접 돌려 보고, 연습문제를 풀어 보기를 바랍니다. 그러면 이미 만들어진 함수형 기능을 단순히 활용하는 것을 넘어 함수형 라이브러리를 만들

어서 제공할 수 있는 프로그래머가 될 수 있을 것입니다.

물론 이 책을 다 본다고 함수형 프로그래밍을 마스터하는 건 아닙니다. 어쩌면 책을 덮은 후부터가 함수형 프로그래밍의 본격적인 시작일 수 있습니다. 실제 프로그램을 만들 때마다 책에서 배운 내용들을 기억하고 어디에 적용할 수 있을지 늘 고민하길 바랍니다. 변경 가능한 객체들을 제거하고, 순수한 함수로 쪼개고, 고차 함수로 일반화하는 과정을 통해서 코드가 점점 간결하고 견고해지는 것을 느낄 수 있을 것입니다.

Q. 책과 독자들에게 기대하는 바가 있다면 알려 주세요.

A. 먼저 이 책이 독자들에게 잘못된 정보를 전달하지 않기를 기원합니다. 그리고 책에서 전달하는 올바른 지식이 독자들의 코드와 개발력에 긍정적인 변화를 주기를 기대합니다. 객체지향 프로그래밍이 그랬던 것처럼, 함수형 프로그래밍이 개발자들의 삶과 문화에 완전히 스며들길 희망합니다. 그래서 더 이상 함수형 프로그래밍을 왜하냐는 질문을 받지 않기를 원합니다. 마지막으로 이 책이 함수형 프로그래밍의 큰 허들인 모나드의 저주를 깰 수 있기를 바랍니다.

책의 구성

이 책은 11개 장으로 구성되어 있고, 각 장에는 개념 설명과 예제코드, 연습문제, 실전응용, 노트가 포함되어 있습니다. 개념 설명 후에는 가급적이면 예제 코드를 통해서 해당 개념을 직접 구현해 봅니다. 예제에서는 자세한 코드 설명을 제공합니다. 모든 본문 예제와 연습 문제는 *https://github.com/funfunStory/fp-kotlin-example*에서 다운로드할 수 있습니다. 함수형의 새로운 개념을 배우는 장에서는 후반부에 '실전응용'이 나옵니다. 실전응용에서는 그 개념을 실전에서 어떻게 활용할 수 있는지에 초점을 맞춘 예제를 다룹니다. 마지막으로 장의 내용을 정리한 '마치며'로 장을 끝냅니다.

1장~3장

1장에서 3장은 책의 전반적인 개념 설명과 예제를 이해하기 위해서 필요한 기본 지식들을 다룹니다. 1장에서는 함수형 프로그래밍을 배우는 이유와 특징, 장점 등을 공부합니다. 2장에서는 본문 예제와 연습문제 풀이를 위한 코틀린 언어에 대해서

알아봅니다. 3장에서는 함수적으로 설계하기 위한 기법인 재귀에 대해서 깊이있게 알아봅니다.

4장~6장

함수형 프로그래밍을 설명하기 위해서 필요한 개념과 함수 들을 소개합니다. 4장에서는 고차 함수를 다루기 위해서 필요한 개념들에 대해서 알아봅니다. 5장에서는 함수형 언어에서 제공하는 여러 가지 고차 함수를 소개하고 직접 구현해 봅니다. 6장에서는 함수적 타입 시스템을 구성하는 다양한 개념에 대해서 설명합니다.

7장~10장

함수형 프로그래밍을 구성하는 주요 컨텍스트이자 대수적 타입인 펑터, 애플리케이티브 펑터, 모노이드, 모나드에 대해서 알아봅니다. 7에서 10장에서는 설명하고자 하는 대수적 타입의 개념을 설명하고, 직접 만들어 보고, 사용해 보고, 검증해 보는 단계를 거칩니다.

11장

함수형 프로그래밍은 예외처리, 로깅, 디버깅, 테스팅을 어떻게 하는지 설명합니다. 이 장은 각 주제마다 구성이 다릅니다. 실용적인 내용을 가장 효율적으로 설명하기에 적합한 방식으로 구성되어 있습니다.

감사의 말

두 해가 넘어 가는 긴 시간 동안 늘 곁에서 응원해 준 은영아 고맙고 사랑해! 책 쓴다고 매일 밤 함께 하지 못한 아빠에게 넘치는 사랑을 준 수아야 고마워! 항상 지친 몸과 다친 마음을 다스려 주시고, 가정에 평안을 주신 하나님께 감사드립니다. 저의 책을 위해서 함께 기도해 주신 구역 식구분들, 책의 품질을 위해서 함께 고민하고 기다려 주신 스터디 멤버, 집필의 기회를 주신 명인 님, 책 쓰느라 피곤한 저 대신 많은 일을 해 주신 MSP팀 여러분 감사드립니다. 마지막으로 말 안 듣는 저를 끝까지 포기하지 않으시고, 책을 완성도 있게 마무리해 주신 선미 님께 깊은 감사의 말씀 전합니다.

조재용

책을 쓰기로 결정한 후 얼마 되지 않아 아내가 임신했고, 지금 그 아이는 벌써 2살이 다 되어 갑니다. 온전히 가족에게 집중해도 모자란 시간이지만, 책을 집필할 수 있도록 배려해 준 아내 애연이와 아들 시원이에게 다시 한번 고맙고 사랑한다고 전하고 싶습니다. 또한 공동 집필에 흔쾌히 응해 준 조재용 님께도 큰 감사를 전하고 싶습니다. 같이 공부하고 리뷰해 주신 FunFunStudy 분들께도 감사드립니다. 끝으로 집필이 처음이라 많이 답답하고 힘들었겠지만 인내심을 가지고 끝까지 신경 써 주신 편집자 문선미 님께 감사를 전하고 싶습니다.

우명인

1장

Functional Programming in Kotlin

함수형 프로그래밍이란?

함수형 프로그래밍의 특징에는 불변성, 참조 투명성, 일급 함수, 게으른 실행 등이 있다. 1장에서는 이 특징들을 명령형 프로그래밍과 비교해 보며 함수형 프로그래밍의 장점을 알아본다. 가장 먼저 순수한 함수란 무엇인지 알아본 후, 부수효과가 없는 프로그램을 어떻게 작성해야 하는지 살펴본다. 이어서 참조 투명성이 프로그램을 더 안전하게 만드는 사례를 살펴본다. 그리고 일급 함수의 개념을 확인한 후, 추상화와 재사용성을 높이기 위해 어떻게 이용하는지 알아본다. 마지막으로 무한 자료구조를 만들면서 게으른 실행이 어떻게 효율적으로 쓰이는지 알아본다. 함수형 프로그래밍으로 가능한 예제들을 이렇게 직접 만들어 보면 함수형 프로그래밍을 해야 하는 이유를 알 수 있을 것이다.

이번 장에서 사용되는 코틀린 코드를 모두 이해해야 하는 건 아니다. 예제를 분석하기 어렵다면 '2장 코틀린으로 함수형 프로그래밍 시작하기'에서 코틀린 문법을 간단히 익히고 시작하자.

1.1 함수형 프로그래밍의 특징

함수형 프로그래밍(functional programming, FP)은 함수를 사용해서 데이터 처리의 참조 투명성을 보장하고, 상태와 가변 데이터 생성을 피하는 프로그래밍 패러다임이다. 객체지향형 프로그래밍(object oriented programming, OOP)의 반대되는 개념이 아니라, 명령형 프로그래밍(imperative programming)과 비교되는 개념으로 보는 게 맞다. 함수형 프로그래밍의 특징은 다음과 같다.

- 불변성(immutable)
- 참조 투명성(referential transparency)
- 일급 함수(first-class-function)
- 게으른 평가(lazy evaluation)

함수형 프로그래밍으로 프로그램을 개발하면 다음과 같이 여러 이점이 있다.

- **부수효과가 없는 프로그램**을 만들 수 있어 동시성 프로그래밍에 적합하다.
- 코드의 복잡도가 낮아 **간결한 코드**를 만들 수 있고, 모듈성이 높아져 유지보수하기 쉽다.
- 프로그램의 예측성을 높여 컴파일러가 **효율적으로 실행되는 코드**를 만들어 준다.

함수형 프로그래밍에 대한 연구는 오래전에 시작되었는데, 그 관심은 요즘 들어 더 커졌다. 최근 개발되는 소프트웨어가 직면한 문제들을 해결하는 데 적합한 특장점들을 지니고 있기 때문이다.

대표적인 순수한 함수형 언어로는 하스켈을 들 수 있다. 그리고 순수한 함수형 언어는 아니지만, 스칼라, 코틀린, 클로저 등도 함수형 특징을 가지고 있다. 또한 자바나 자바스크립트 등 함수형 특징들을 포함하지 않던 언어에서도 함수형 프로그래밍을 지원하는 버전이 속속 출시되었다. 이 책에서는 JVM에서 실행되고 객체지향 프로그래밍과 함수형 프로그래밍을 모두 지원하는 멀티 패러다임 언어, 코틀린으로 함수형 프로그래밍을 설명한다. 코틀린에 익숙하지 않다면 2장을 먼저 살펴봐도 된다.

1.2 순수한 함수란 무엇인가?

순수한 함수란 무엇인지 살펴보기 전에 수학에서의 함수 개념을 먼저 떠올려 보자. 수학에서 함수 y = f(x)는 어떤 입력값 x에 대해서 항상 동일한 결괏값 y를 출력한다. 예를 들어 더하기 함수는 1 + 1의 결괏값으로 항상 2를 반환한다. 이를 프로그래밍적으로 모델링한 것이 순수한 함수이다. 코드로는 다음과 같이 표현한다.

코드 1-1 순수한 함수의 예

```
fun pureFunction(x: Int, y: Int): Int = x + y
```

이러한 순수한 함수에는 다음과 같은 특징이 있다.

- 동일한 입력으로 실행하면 항상 동일한 결과가 나온다.
- 부수효과가 없다.

이 두 특징을 조금 더 자세히 알아보자.

동일 입력 동일 출력

순수한 함수의 첫 번째 특징은 동일한 입력에 항상 동일한 결과를 돌려준다는 점이다.

그렇다면 어떤 경우에 똑같은 입력에도 다른 결과를 돌려줄까? 전역 변수, 파일, 네트워크 등으로부터 데이터를 가져올 때다. 간단한 예를 하나 보자.

코드 1-2 순수하지 못한 함수의 예 1 - 외부 변수 참조

```
fun main(args: Array<String>) {
    println(impureFunction(1, 2))   // "13" 출력
    z = 20
    println(impureFunction(1, 2))   // "23" 출력
}

var z = 10

// 순수하지 않은 함수
fun impureFunction(x: Int, y: Int): Int = x + y + z
```

`impureFunction` 함수는 위치에 따라 같은 값을 입력해도 다른 결괏값을 낸다. 함수 내부에서 외부 변수 `z`를 참조해서 값을 생성하기 때문이다. 따라서 `impureFunction` 함수가 어떤 결괏값을 낼지 예측하기 어렵다.

부수효과 없는 코드

순수한 함수의 두 번째 특징은 부수효과가 없다는 것이다. 여기서 부수효과란 함수가 실행되는 과정에서 외부의 상태(데이터)를 사용 또는 수정하는 걸 말한다. 전역 변수나 정적 변수를 수정하거나, 파일이나 네트워크를 출력하는 작업 등이 부수효

과에 해당하며, 예외 발생도 부수효과에 속한다. 다음은 외부 변수를 수정하여 부수효과를 일으키는 함수의 예다.

코드 1-3 순수하지 못한 함수의 예 2 - 외부 변수 수정

```
var z = 10

// 부수효과가 있는 함수
fun impureFunctionWithSideEffect(x: Int, y: Int): Int {
    z = y
    return x + y
}
```

`impureFunctionWithSideEffect` 함수는 동일 입력에 매번 동일한 결과를 돌려주지만, 외부 변수인 z의 값을 수정하기 때문에 순수한 함수가 아니다.

순수한 함수의 효과와 그 외 고려사항

함수형 프로그래밍은 동일한 입력에 대해 동일한 결과를 반환하는 특성 때문에 결과에 대한 추론이 가능하고 테스트도 쉽다. 또한 컴파일 타임에 코드를 최적화하거나 오류 코드를 예측하고 경고하는 등 많은 것을 할 수 있다. 또한 동시성 프로그래밍에서는 공유 자원이 변경될 걱정 없이 더 안전한 프로그램을 만들 수 있다. 그리고 순수한 함수의 특성 덕에 참조 투명성(referential transparency)도 만족하게 된다(참조 투명성은 1.4절에서 알아본다).

반면 순수하지 못한 함수는 어떤 결괏값을 낼지 예측하기 어렵다. 눈에 잘 띄지 않는 부수효과를 남기므로 테스트하기도 어렵다. 순수하지 못한 함수를 사용하는 함수 역시 순수하지 못한 함수가 되므로 주의해야 한다.

그렇다면 함수형 프로그래밍에서는 파일 입출력이나 네트워크 통신 등의 작업을 해서는 안 되는 것일까? 어느 정도 규모의 프로그램을 만들어 봤다면 이런 제약은 상상하기 어려울 것이다. 순수한 함수형이 아닌 하이브리드 언어들은 이런 작업을 허용한다.

순수한 함수형 언어인 하스켈에서는 순수하지 못한 작업을 언어 차원에서 완전히 분리하는 방법으로 언어 순수성을 유지한다. 이러한 작업들에 대한 함수적 해법은 순수하지 못한 함수의 선언을 최소화하고, 순수하지 못한 작업이 필요한 부분만 모듈화하여 분리하는 방식으로 접근하는 것이다.

1.3 부수효과 없는 프로그램 작성하기

부수효과는 함수의 반환값이 아닌, 외부의 상태에 영향을 미치는 것을 말한다. 부수효과를 만드는 예를 더 살펴보자.

공유 변수 수정으로 인한 부수효과

함수 내에서 전역 변수와 같은 공유 변수를 수정하면 부수효과가 발생하고, 이 변수를 참조하는 변수는 결과가 외부 요인에 의해 달라진다.

코드 1-4 공유 변수 수정에 의한 부수효과

```
fun main(args: Array<String>) {
    println(impureFunction(1, 2))     // "13" 출력
    println(withSideEffect(10, 20))   // "30" 출력
    println(impureFunction(1, 2))     // "23" 출력
}

var z = 10

// 순수하지 않은 함수
fun impureFunction(x: Int, y: Int): Int = x + y + z

// 부수효과가 있는 함수
fun withSideEffect(x: Int, y: Int): Int {
    z = y
    return x + y
}
```

`withSideEffect` 함수는 함수의 결괏값인 `x + y`와 관계없는 외부 변수 `z`의 값을 변경하였다. `withSideEffect` 함수의 부수효과 때문에 `impureFunction` 함수의 결괏값이 달라졌다. 이러한 결괏값의 변화는 프로그래머가 예측하지 못한 것일 수 있으며, 예측했더라도 이런 식으로 외부 변수를 변경하는 건 좋지 않다. 또한 `z`의 상태를 예측하기 어렵기 때문에 `z`를 사용하는 다른 함수에서 예외처리가 필요할 수 있다. 이러한 불확실성은 많은 노이즈 코드(boilerplate code)를 생성하고 프로그램을 복잡하게 만든다. 이처럼 부수효과는 디버깅과 테스트를 어렵게 하고, 버그를 만들기 쉽게 한다.

객체의 상태 변경으로 인한 부수효과

객체의 상태를 변경하는 것 역시 부수효과를 일으킨다.

코드 1-5 객체의 상태 변경에 의한 부수효과

```
data class MutablePerson(var name: String, var age: Int)

// 인자로 들어온 객체의 상태를 변경
fun addAge(person: MutablePerson, num: Int) {
    person.age += num
}
```

addAge 함수에서는 매개변수로 받은 MutablePerson 객체를 수정하고 있다. 만약 다른 함수나 모듈에서 동일 인스턴스를 참조하면 이 부수효과의 영향을 받을 수 있다. 코드 1-5에서 MutablePerson 객체는 수정 가능한 가변(mutable) 객체다. 함수형 프로그래밍에서는 객체를 만들 때 수정 불가능한 불변(immutable) 객체로 만들어야 한다. 그런데 MutablePerson을 불변 객체로 만들면 컴파일 오류가 발생한다. 함수형 프로그래밍에서는 이 문제를 해결하기 위해서 객체를 수정하는 대신 새로운 객체를 생성한다. 코드 1-5의 MutablePerson을 불변 객체로 만들고, addAge 함수를 새로 작성해 보자.

코드 1-6 불변 객체로 addAge 함수 작성하기

```
// 객체의 속성을 val로 선언하면 수정이 불가능함
data class ImmutablePerson(val name: String, val age: Int)

// Person 객체를 수정하지 않고, 새로운 객체를 생성하여 반환
fun addAge(person: ImmutablePerson, num: Int): ImmutablePerson {
    return ImmutablePerson(person.name, person.age + num)
}
```

MutablePerson 객체를 수정이 불가능하도록 ImmutablePerson으로 변경하였다. 그리고 addAge 함수에서 ImmutablePerson을 수정하지 않고, age가 증가한 새로운 ImmutablePerson 객체를 만들어서 돌려준다. ImmutablePerson 객체가 새로운 인스턴스로 생성되었기 때문에 기존에 매개변수로 받은 ImmutablePerson 객체에는 영향을 주지 않는다. 따라서 부수효과는 사라졌다.

이상으로 부수효과를 없애는 간단한 함수적 해법을 확인했다. 하지만 실전에서는 이처럼 간단하지 않을 수도 있다. ImmutablePerson 객체의 나이가 정말 증가했다면, 다음에 ImmutablePerson 객체를 참조할 때도 나이가 증가된 상태의 ImmutablePerson 객체를 얻어야 한다. 즉, 값의 수정에 영속성이 있어야 한다. 이 작업을 위해서는 수정된 ImmutablePerson을 데이터베이스에 넣어야 하고, 이것은

또다시 부수효과를 일으킨다.

불필요한 부수효과를 최소화하는 것이 함수적 해법이다. 부수효과를 수반해야만 하는 작업은 반드시 순수한 영역과 분리한다. 그리고 분리된 영역(부수효과가 발생하는 영역)이 외부로 드러나지 않도록 설계해야 한다. 이 외에도 유의해야 할 사항은 많다. 함수적으로 프로그램을 설계하고 작성하는 방법을 차근차근 익혀보자.

1.4 참조 투명성으로 프로그램을 더 안전하게 만들기

순수한 함수는 참조 투명성(referential transparency)을 만족시킨다. 참조 투명성이란, **프로그램의 변경 없이 어떤 표현식을 값으로 대체할 수 있다**는 뜻이다. 수학의 개념을 예로 들면 1 + 1은 값 2로 대체할 수 있다. 참조에 투명한 함수 f를 평가하면 동일한 입력에 대해서 동일한 결과를 돌려준다. 이때 함수 f는 순수한 함수이다. 순수한 함수 f의 표현식 $f(x)$가 y를 반환한다면, $f(x)$는 y로 대체될 수 있다.

참조 투명성은 프로그래머나 컴파일러가 평가 결과를 추론할 수 있게 한다. 그래서 프로그램이 실행되기 전에 컴파일러가 코드를 최적화하거나 코드가 평가되는 시점을 늦출 수 있다. 코드에 예외가 사라져서 간결해지고 버그가 발생할 가능성은 낮아진다. 멀티스레드 코드에서도 스레드 안전성(thread safety)에 대한 고민을 덜 수 있다.

참조 투명하지 않은 함수

참조 투명하지 않은 함수를 예로 들어 보자. 다음은 이름을 입력받아서 인사말을 반환하는 함수이다.

코드 1-7 전역 변수를 참조하는 Hello 함수

```
var someName: String = "Joe"

fun hello1() {
    println("Hello $someName")
}
```

`hello1` 함수는 외부 변수를 참조하여 출력하고 있기 때문에 참조에 투명하지 않다. `hello1` 함수를 호출할 때 결과는 전역 변수의 값에 따라 달라질 것이다. 전역 변수를 참조하지 않도록 코드 1-7을 수정해 보자.

코드 1-8 매개변수를 입력받는 Hello 함수

```
fun hello2(name: String) {
    println("Hello $name")
}
```

hello2 함수는 전역 변수를 참조하지 않고, 값을 매개변수로 받았다. 이제는 동일한 입력에 동일한 출력을 한다. 하지만 콘솔에 출력하는 작업 자체가 부수효과를 일으키기 때문에 여전히 참조에 투명하다고 보기 어렵다.

참조 투명한 함수

코드 1-8을 참조 투명하게 수정해 보자.

코드 1-9 참조 투명한 Hello 함수

```
fun main(args: Array<String>) {
    val result = transparent("Joe")
    print(result)
}

fun transparent(name: String): String {
    return "Hello $name"
}

fun print(helloStr: String) {
    println(helloStr)
}
```

hello2 함수에서 transparent 함수를 분리하였다. transparent 함수는 참조에 투명하다. 언제 어디서 호출해도 문제가 없기 때문에 재사용성이 높고, 테스트하기 쉽다. print 함수가 여전히 부수효과를 일으키지만, '인사말을 화면에 출력하라'는 프로그램 요구사항을 만족시키기 위해서 불가피한 부분이다.

아주 간단한 예를 들었기에 부수효과와 순수한 영역을 이렇게까지 분리해야 되나 하는 의문이 들 수도 있다. 하지만 이는 함수형 프로그래밍 관점에서 올바른 설계 방향이다. 부수효과를 일으키는 영역과 순수한 영역을 되도록 분리하고, 참조에 투명한 함수들로 구성하는 것이 좋다. 이렇게 하면 코드에 버그가 생길 확률이 줄어들고 더 안전한 프로그램을 작성할 수 있다.

1.5 일급 함수란?

함수형 프로그래밍에 관심이 있다면 일급 함수라는 단어는 어디선가 들어 봤을 것이다. 일급 함수란 무엇을 말하는 걸까? 일급 함수가 무엇인지 알아보기 위해 우선 일급 객체는 무엇인지 알아보고 시작하자.

일급 객체(first-class object)

자바를 비롯한 대부분의 객체지향 언어는 일급 객체를 지원한다. 일급 객체는 다음 세 가지 조건을 만족시키는 객체를 의미한다.

- 객체를 함수의 매개변수로 넘길 수 있다.
- 객체를 함수의 반환값으로 돌려 줄 수 있다.
- 객체를 변수나 자료구조에 담을 수 있다.

예를 들어 코틀린의 최상위 객체인 Any는 일급 객체이다. 따라서 다음과 같이 선언하여 사용할 수 있다.

코드 1-10 일급 객체의 조건을 만족하는 예

```
// Any를 함수의 매개변수로 넘길 수 있다.
fun doSomethingWithAny(any: Any) {
    // do something
}

// Any를 함수의 반환값으로 돌려 줄 수 있다.
fun doSomethingWithAny(): Any {
    return Any()
}

// Any를 List 자료구조에 담을 수 있다.
var anyList: List<Any> = listOf(Any())
```

일급 함수(first-class function)

동일한 조건들을 함수에 적용했을 때, 다음 조건들을 만족하면 함수는 일급 함수라고 할 수있다.

- 함수를 함수의 매개변수로 넘길 수 있다.
- 함수를 함수의 반환값으로 돌려 줄 수 있다.

- 함수를 변수나 자료구조에 담을 수 있다.

일급 함수의 조건을 만족하는 함수로 예제를 작성해 보자.

코드 1-11 일급 함수의 조건을 만족하는 예

```
// 함수를 함수의 매개변수로 넘길 수 있다.
fun doSomething(func: (Int) -> String) {
    // do something
}

// 함수를 함수의 반환값으로 돌려 줄 수 있다.
fun doSomething(): (Int) -> String {
    return { value -> value.toString() }
}

// 함수를 List 자료구조에 담을 수 있다.
var funcList: List<(Int) -> String> = listOf({ value -> value.toString() })
```

함수형 프로그래밍은 일급 함수에서부터 시작한다고 봐도 과언이 아니다. 일급 함수를 통해서 더 높은 추상화가 가능하고, 코드의 재사용성을 높일 수 있다. 뒤에서 설명하는 람다식, 고차 함수, 커링, 모나드 등의 함수적 개념들도 기본적으로 일급 함수가 아니면 존재할 수 없다.

1.6 일급 함수를 이용한 추상화와 재사용성 높이기

일급 함수를 활용하면 명령형 프로그래밍이나 객체지향 프로그래밍에서 할 수 없는 추상화가 가능하다. 이번 절에서는 간단한 계산기를 명령형으로 만들어 보고, 그 코드를 객체지향과 함수형으로 리팩터링한다. 그 과정에서 일급 함수가 추상화와 재사용성을 어떻게 높이는지 살펴본다.

간단한 계산기 예제

우선 덧셈과 뺄셈만 할 수 있는 아주 간단한 `SimpleCalculator` 계산기를 명령형으로 구현해 보자.

코드 1-12 명령형 프로그래밍으로 만든 계산기

```
fun main(args: Array<String>) {
    val calculator = SimpleCalculator()
```