Chapter10 고차함수: 람다를 파라미터와 반환값으로 사용

10장에서 다루는 내용

- 함수 타입
- 고차 함수와 코드를 구조화할 때 고차 함수를 사용하는 방법
- 인라인 함수
- 비로컬 return과 레이블
- 익명 함수

10장애서는 람다를 인자로 받거나 반환하는 함수인 고차 함수를 만드는 방법을 다룬다. 고차 함수로 코드를 더 간결하게 다듬고 코드 중복을 없애고 더 나은 추상화를 구축하는 방법을

살펴본다.

또한 람다를 사용함에 따라 발생할 수 있는 성능상 부가 비용을 없애고 람다 안에서 더 유연하게 흐름을 제어할 수 있는 코틀린 특성인 인라인 함수를 설명한다.

고차 함수: 다른 함수를 인자로 받거나 반환하는 함수 정의

고차 함수

- 다른 함수를 인자로 받거나 함수를 반환하는 함수
- 코틀린에서는 람다나 함수 참조를 사용해 함수를 값으로 표현할 수 있다.
- 따라서 고차함수는 람다나 함수 참조를 인자로 넘길 수 있거나 람다나 함수 참조를 반환하는 함수
    - 함수를 인자를 받는 동시 함수를 반환하는 함수도 고차 함수
    - eg) filter

함수타입 선언: 함수 타입은 람다의 파라미터 타입과 반환 타입을 지정한다.

```
(Int, String) → Unit
// 파라미터 타입   // 반환 타입
```

- 함수 타입을 정의하려면 함수 파라미터의 타입을 괄호 안에 넣고 그 뒤에 화살표를 추가한 뒤, 함수의 반환 타입을 지정한다.
- 함수 타입을 선언할 때는 반환 타입을 반드시 명시해야 한다.

```
val sum: (Int, Int) → Int = (x, y → x + y)
val canReturnNull: (Int, Int) → Int = (x, y → null)
val funOrNull: ((Int, Int) → Int)? = null
```

- 람다식 안에서 x와 y의 타입을 생략해도 되는점을 유의하자
- 변수 선언의 일부분인 함수 타입 안에 파라미터 타입을 지정했기 때문에 람다 자체에는 파라미터 타입을 굳이 지정할 필요가 없다.

- canReturnNull과 funOrNull에는 큰 차이가 있다.

인자로 전달 받은 함수 호출

```
fun twoAndThree(operation: (Int, Int) -> Int) { // 함수 타입인 파라미터 선언
    val result = operation(2, 3) // 함수 타입인 파라미터 호출
    println("The result is $result")
}

fun main() {
    twoAndThree { a, b -> a + b }
    // The result is 5
    twoAndThree { a, b -> a * b }
    // The result is 6
}
```

- 인자로 받은 함수를 호출하는 구문은 일반 함수를 호출하는 구문과 같다.

```
fun twoAndThree(
    operation: (operandA: Int, operandB: Int) -> Int,
) {
    val result = operation(2, 3)
    println("The result is $result")
}

fun main() {
    twoAndThree { operandA, operandB -> operandA + operandB }
    // The result is 5
    twoAndThree { alpha, beta -> alpha + beta }
    // The result is 5
}
```

- 함수 타입에서 파라미터 이름을 지정할 수도 있으나 원하는 다른 이름 붙여도 된다.
- 파라미터 이름은 타입 검사 시 무시 된다.
    - 하지만 가독성이 좋아지고 IDE는 이름을 코드 완성에 사용할 수 있다.

```
fun String.filter(predicate: (Char) → Boolean): String
//  수신 객체 타입   파라미터 이름     파라미터 함수 타입
```

- filter 함수는 술어를 파라미터로 받는다.
- predicate 파라미터는 문자를 파라미터로 받고 불리언 결괏값을 반환
- 술어는 filter

```
fun String.filter(predicate: (Char) → Boolean): String {
    return buildString {
        for (char in this@filter) {
            if (predicate(char)) append(char)
        }
    }
}

fun main() {
    println("ab1c".filter { it in 'a'..'z' })
    // abc
}
```

- 확장 함수와 buldString 함수 모두 수신 객체를 정의하기 때문에 this 앞에 레이블을 붙여 buildString 람다의 수신 객체(StringBuilder 인스턴스)가 아니라 filter의 바깥쪽 수신 객체(String)에 접근해야 한다.

자바에서 코틀린 함수 타입 사용

- 5장에서 본 것 처럼 자동 SAM 변환을 통해 코틀린 람다를 함수형 인터페이스를 요구하는 자바 메서드에게 넘길 수 있다.
- 이는 코틀린 코드가 자바 라이브러리에 의존할 수 있고, 자바에서 정의된 고차 함수를 아무 문제없이 사용할 수 있다는 의미
- 마찬가지로 함수 타입을 사용하는 코틀린 코드도 자바에서 쉽게 호출할 수 있다.

```
// 코틀린 선언
fun processTheAnswer(f: (Int) -> Int) {
    println(f(42))
}
```

```
// 자바 호출
processTheAnswer(number -> number + 1);
```

- 자바 람다는 자동으로 코틀린 함수 타입으로 변환된다.
- 자바에서 람다를 인자로 받는 코틀린 표준 라이브러리의 확장 함수를 쉽게 사용 가능하다
    - 하지만 첫번째 인자로 수신객체를 명시적으로 전달해야 한다.

함수 타입: 자세한 구현

- 내부에서 코틀린 함수 타입은 일반 인터페이스다.
- 함수 타입의 변수는 FunctionN 인터페이스를 구현한다.
    - 즉 해당 인터페이스를 구현하는 인스턴스
- 각 인터페이스에는 invoke 라는 유일한 메서드가 정의돼 있다.
    - 이 메서드를 호출하면 함수가 호출된다.
- FunctionN 인터페이스는 컴파일러가 생성한 합성 타입이다.
    - 따라서 표준 라이브러리에서 정의를 찾을 수 없다.
    - 이는 파라미터의 개수 제한 없이 원하는 만큼 함수에 대한 인터페이스 사용이 가능하다는 뜻

함수 타입의 파라미터에 대해 기본값을 지정할 수 있고, 널이 될 수도 있다.

```
fun <T> Collection<T>.joinToString(
    separator: String = ", ",
    prefix: String = "",
```

```
    postfix: String = ""
): String {
    val result = StringBuilder(prefix)

    for ((index, element) in this.withIndex()) {
        if (index > 0) result.append(separator) // 기본 toString 메서드를 사용해 객체
        result.append(element)
    }

    result.append(postfix)
    return result.toString()
}
```

- 파라미터를 함수 타입으로 선언할 때도 기본값을 지정할 수 있다.
- 항상 객체를 toString 메서드를 통해 문자열로 바꾼다.
    - 충분하지 않다면?
- 원소를 문자열로 바꾸는 방법을 람다로 전달
- 하지만 joinToString 호출할 때마다 매번 람다 넘기면 기본 동작으로 충분한 경우 함수 호출이 더 불편해진다.
    - 함수 타입의 파라미터에 대한 기본값으로 람다식 넣기

```
fun <T> Collection<T>.joinToString(
    separator: String = ", ",
    prefix: String = "",
    postfix: String = "",
    transform: (T) -> String = { it.toString() }, // 함수 타입 파라미터 및 람다를 기본값
): String {
    val result = StringBuilder(prefix)

    for ((index, element) in this.withIndex()) {
        if (index > 0) result.append(separator)
        result.append(transform(element)) // tranform 파라미터 인자 함수 호출
    }

    result.append(postfix)
```

```
    return result.toString()
}

fun main() {
    val letters = listOf("Alpha", "Beta")
    println(letters.joinToString()) // 기본 변환 함수
    // Alpha, Beta
    println(letters.joinToString { it.lowercase() }) // 람다를 인자로 전달
    // alpha, beta
    println(
        letters.joinToString(separator = "! ", postfix = "! ",
            transform = { it.uppercase() })
    )
    // ALPHA! BETA!
}
```

```
fun <T> Collection<T>.joinToString(
    separator: String = ", ",
    prefix: String = "",
    postfix: String = "",
    transform: ((T) -> String)? = null,
): String {
    val result = StringBuilder(prefix)
    for ((index, element) in this.withIndex()) {
        if (index > 0) result.append(separator)
        val str = transform?.invoke(element)
            ?: element.toString()
        result.append(str)
    }

    result.append(postfix)
    return result.toString()
}
```

- invoke 를 활용하면 더 짧게 만들수 있다.

함수를 함수에서 반환

```
enum class Delivery { STANDARD, EXPEDITED }

class Order(val itemCount: Int)

fun getShippingCostCalculator(delivery: Delivery): (Order) -> Double {
    if (delivery == Delivery.EXPEDITED) {
        return { order -> 6 + 2.1 * order.itemCount }
    }

    return { order -> 1.2 * order.itemCount }
}

fun main() {
    val calculator = getShippingCostCalculator(Delivery.EXPEDITED)
    println("Shipping costs ${calculator(Order(3))}")
    // Shipping costs 12.3
}
```

- 함수에서 함수를 반환하는 경우는 적지만 유용하다.
    - eg) 프로그램의 상태나 다른 조건에 따라 달라질 수 있는 로직이 있다구 생각해보자.
- 함수를 반환하려는 함수를 정의하려면 함수의 반환 타입으로 함수 타입을 지정한다.

```
data class Person(
    val firstName: String,
    val lastName: String,
    val phoneNumber: String?,
)

class ContactListFilters {
    var prefix: String = ""
    var onlyWithPhoneNumber: Boolean = false

    fun getPredicate(): (Person) -> Boolean {
```

```
        val startsWithPrefix = { p: Person ->
            p.firstName.startsWith(prefix) || p.lastName.startsWith(prefix)
        }
        if (!onlyWithPhoneNumber) { // 함수 타입의 변수 반환
            return startsWithPrefix
        }
        return {
            startsWithPrefix(it)
                    && it.phoneNumber != null // 람다를 반환
        }
    }
}

fun main() {
    val contacts = listOf(
        Person("Dmitry", "Jemerov", "123-4567"),
        Person("Svetlana", "Isakova", null)
    )
    val contactListFilters = ContactListFilters()
    with(contactListFilters) {
        prefix = "Dm"
        onlyWithPhoneNumber = true
    }
    println(
        contacts.filter(contactListFilters.getPredicate())
    )
    // [Person(firstName=Dmitry, lastName=Jemerov, phoneNumber=123-4567
}
```

- eg) UI의 상태에 따라 어떤 연락처 정보를 표시할지 결정해야 하는 경우

```
fun main() {
        val averageWindowsDuration = log
```

```
    .filter { it.os in setOf(OS.IOS, OS.ANDROID) }
    .map(SiteVisit::duration)
    .average()

    println(averageWindowsDuration)
}
```

- 복잡하게 하드코딩된 필터를 사용해 방문 데이터 분석하기
- 이럴 때 람다가 유용하다.
    - 함수 타입을 사용하면 필요한 조건을 파라미터로 뽑아낼 수 있다.

```
fun List<SiteVisit>.averageDurationFor(predicate: (SiteVisit) → Boolean) =
    filter { predicate }.map(SiteVisit::duration).average()

fun main() {
    println(log.averageDurationFor { it.os in setOf(OS.IOS, OS.ANDROID) })
    println(log.averageDurationFor { it.os == OS.IOS && it.path == "/signup"} )
}
```

- 코드 중복을 줄일 때 함수 타입이 상당히 도움이 된다.
- 코드의 일부분을 복사해 붙여 넣고 싶은 경우가 있다면 그 코드를 람다로 만들면 중복을 제거할 수 있을 것이다.
- 람다를 사용하면 데이터의 반복을 추출할 수 있을 뿐 아니라 반복적인 행동도 추출할 수 있다.

인라인 함수를 사용해 람다의 부가 비용 없애기

- 5장에서 코틀린이 보통 람다를 익명 클래스로 컴파일 한다고 설명, 이것은 람다식마다 새로운 클래스가 생기고 람다가 변수를 캡쳐한 경우 람다 정의가 포함된 코드를 호출하는 시점마다 새로운 객체가 생긴다는 뜻
    - 이로인해 부가비용
- 반복되는 코드를 별도의 라이브러리 함수로 빼내되 직접 실행될 때만큼 효율적인 코드를 컴파일러가 생성하게 만들 수 없을까?

- inline 변경자 사용
- 컴파일러는 함수 호출을 생성하는 대신에 함수를 구현하는 코드로 바꿔치기 해준다.

인라이닝이 작동하는 방식

```
inline fun <T> synchronized(lock: Lock, action: () → T): T {
    lock.lock()
    try {
        return action()
    }
    finally {
        lock.unlock()
    }

}

fun main() {
    val l = ReentrantLock()
    synchronized(l) {
        // ...
    }
}
```

- 어떤 함수를 inline으로 선언하면 그 함수의 본문이 인라인된다.
  - 함수를 호출하는 코드를 함수를 호출하는 바이트코드 대신에 함수 본문을 번역한 바이트 코드로 컴파일 한다.
- inline으로 선언해주었기 때문에 호출하는 코드는 모두 자바의 synchronized 문과 같아진다

```
fun foo(l: Lock) {
    println("Before critical section")
    synchronized(l) {
        println("Critical section")
```

```
    }
    println("After critical section")
}
```

- 위에서 inline으로 정의한 synchronized를 위처럼 사용했을 때

```
fun __foo__(l: Lock) {
    println("Before critical section")
    lock.lock()    // synchronized 함수 인라이닝
    return try {   // synchronized 함수 인라이닝
        println("Critical section") // 람다 코드의 본문이 인라이닝된 코드
    } finally {    // synchronized 함수 인라이닝
        lock.unlock() // synchronized 함수 인라이닝
    }
    println("Outside critical section")
}
```

- 실제로 컴파일된 바이트코드는 다음과 같은 형식이다.
- synchronized 함수의 본문뿐 아니라 synchronized에 전달된 람다의 본문도 함께 인라인되는 점에 주목하자

```
class LockOwner(val lock: Lock) {
    fun runUnderLock(body: () -> Unit) {
        synchronized(lock, body) // 람다 대신 함수 타입인 변수를 인자로 넘긴다.
    }
}
```

- 인라인 함수를 호출하면서 람다를 넘기는 대신에 함수 타입의 변수를 넘길 수도 있다.
- 이런 경우 람다 본문은 인라이닝 되지 않는다.
    - 인라인 함수를 호출하는 코드 위치에서는 변수에 저장된 람다의 코드를 알 수 없다.
    - 대신 synchronized의 함수의 본문만 인라이닝된다.

```
class LockOwner(val lock: Lock) {
    fun __runUnderLock__(body: () -> Unit) {
```

```
        lock.lock()
        try {
            body()   // synchronized를 호출하는 부분에서 람다를 알 수 없으므로 인라이
        } finally {
            lock.unlock()
        }
    }
}
```

- 실제로 컴파일된 바이트코드는 다음과 같은 형식이다.
- 하나의 인라인 함수를 두 곳에서 각각 다른 람다를 사용해 호출한다면 그 두 호출은 각각 따로 인라이닝된다.
- 인라인 함수의 본문 코드가 호출 지점에 복사되고 각 람다의 본문이 인라인 함수의 본문 코드에서 람다를 사용하는 위치에 복사된다
- 함수에 더해 프로퍼티 접근자에도 inline을 붙일 수 있다.
    - 코틀린을 reified generic, 실체화한 제네릭에서 이런 기능이 유용하다.

인라인 함수의 제약

- 인라이닝 방식으로 인해 람다를 사용하는 모든 함수를 인라이닝할 수는 없다.
- 함수가 인라이닝될 때 그 함수에 인자로 전달된 람다식의 본문은 결과 코드에 직접 들어갈 수 있다.
- 하지만 함수가 파라미터로 전달받은 람다를 본문에 사용하는 방식이 한정될 수 밖에 없다.
    - 파라미터로 받은 람다를 다른 변수에 저장하고 나중에 그 변수를 사용할 때에는 람다를 표현하는 객체가 어딘가는 존재해야 해서 람다를 인라이닝할 수 없다

```
class FunctionStorage {
    var myStoredFunction: ((Int) -> Unit)? = null
    inline fun storeFunction(f: (Int) -> Unit) {
        myStoredFunction = f  // 전달된 파라미터를 저장한다.
                              // 따라서 컴파일러는 모든 호출 지점에서 이 코드를 대치
```

```
    }
}
```

- `Illegal usage of inline parameter` 인라이닝 금지시킨다.

```
public fun <T, R> Sequence<T>.map(transform: (T) -> R): Sequence<R> {
    return TransformingSequence(this, transform)
}
```

- 시퀀스는 람다를 받아 모든 시퀀스 원소에 적용해 새 시퀀스를 반환하는 함수가 많다
- map 함수는 `transform` 파라미터로 전달받은 함수 값을 호출하지 않는 대신 `TransformingSequence` 클래스의 생성자로 넘기면 생성자에서 전달받은 람다를 프로퍼티로 저장한다.
- 이런 기능을 지원하려면 map에 전달되는 transform 인자를 일반적인 함수표현, 즉 함수 인터페이스를 구현하는 익명 클래스 인스턴스로 만들 수 밖에 없다.
- 둘 이상의 람다를 인자로 받는 함수에서 일부 람다만 인라이닝 하고 싶을 때
    - ex) 어떤 람다에 너무 많은 코드가 들어가거나 어떤 람다에 인라이닝하면 안 되는 코드가 들어가는 경우
    - `noinline` 키워드를 파라미터 이름 앞에 붙여 인라이닝을 금지할 수 있다

컬렉션 연산 인라이닝

- 코틀린 표준 라이브러리의 컬렉션 함수는 대부분 람다를 인자로 받는다.
- 성능은 어떨까?
    -

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

fun main() {
```

```
    println(people.filter { it.age < 30 })
    // [Person(name=Alice, age=29)]
}
```

- filter 함수는 인라인 함수다.
- 따라서 컬렉션에서 직접 거르는것과 람다를 사용해 거르는 것과 바이트코드는 거의 비슷하다.

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

fun main() {
    println(
        people.filter { it.age > 30 }
            .map(Person::name)
    )
    // [Bob]
}
```

- filter와 map을 연쇄해서 사용하는 경우 리스트를 걸러낸 결과를 저장하는 중간 리스트를 만든다.
- asSequence를 통해 리스트 대신 시퀀스를 사용하면 중간 리스트로 인한 부가 비용은 줄어든다.
    - 하지만 시퀀스에 사용된 람다는 인라이닝되지 않는다.
    - 각 중간 시퀀스는 람다를 필드에 저장하는 객체로 표현되며, 최종 연산은 중간 시퀀스에 있는 여러 람다를 연쇄 호출하기 때문이다.
- 따라서 지연 계산을 통해 성능 향샹시키려는 이유로 모든 컬렉션 연산에 시퀀스를 사용하려고 하면 안된다.
    - 컬렉션의 크기가 큰 경우가 아니라면

언제 함수를 인라인으로 선언할지 결정

- 코드 여기저기에 인라인을 사용하는 것은 좋지 않다.

- 일반 함수 호출의 경우 JVM은 이미 바이트코드에서 기계어 번역 과정(JIT)에서 가장 효율적인 방향으로 호출을 인라이닝해준다.
- 반면 람다를 인자로 받는 함수를 인라이닝하면 이익이 더 많다.
  - 장점
    - 함수 호출 비용을 줄이며, 람다를 표현하는 클래스와 람다 인스턴스에 해당하는 객체를 만들 필요가 없다.
    - 현재의 jvm은 함수 호출과 람다를 인라이닝해줄 정도로 똑똑하지는 못한다.
    - 일반 람다에서는 사용할 수 없는 몇가지 기능을 사용할 수 있다.
      - 비로컬 return
  - 단점
    - 코드 크기에 주의해야한다., 바이트코드가 전체적으로 아주 커질 수 있다.
    - 그런 경우 람다인자와 무관한 코드를 비인라인 함수로 빼낼 수 있다.
    - 즉 비인라인 함수는 크기가 작게 관리하는 것이 좋다.

람다에서 반환

- 람다안의 return은 어떻게 동작할까?

람다 안의 return 문: 람다를 둘러싼 함수에서 반환

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

fun lookForAlice(people: List<Person>) {
    for (person in people) {
        if (person.name == "Alice") {
            println("Found!")
            return
        }
    }
    println("Alice is not found")
```

```
}

fun main() {
    lookForAlice(people)
    // Found!
}
```

- forEach로 바꿔써도 될까? 그렇다.

```
val people = listOf(Person("Alice", 29), Person("Bob", 31))

fun lookForAlice(people: List<Person>) {
    people.forEach {
        if (it.name == "Alice") {
            println("Found!")
            return
        }
    }
    println("Alice is not found")
}
```

- 람다 안에서 리턴을 사용하면 람다에서만 반환되는 것이 아니라 그 람다를 호출하는 함수가 실행을 끝내고 반환된다.
- 그렇게 자신을 둘러싸고 있는 블록보다 더 바깥에 있는 다른 블록을 반환하게 만드는 리턴문을 비로컬 리턴문이라고 부른다.
    - 마치 자바의 for문 안에서나 synchronized 블록 안에서 return 사용 시 블록을 끝내지 않고 메서드를 반환시키는 것과 동일하다
- 이렇게 리턴이 바깥쪽 함수를 return으로 반환시킬 수 있으려면 람다를 인자로 받는 함수가 인라인 함수인 경우 뿐이다.
    - 인라이닝되지 않는 함수는 람다를 변수에 저장될 수 있고, 그런 경우 함수가 반환된 다음에 나중에 람다를 실행할 수도 있기 때문에 원래 함수를 반환시키기 늦은 시점이 되어버린다

람다로부터 반환: 레이블을 사용한 return

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

fun lookForAlice(people: List<Person>) {
    people.forEach label@{ // 람다식 앞에 레이블
        if (it.name != "Alice") return@label // 앞에서 정의한 레이블 참조
        print("Found Alice!")
    }
}

fun main() {
    lookForAlice(people)
    // Found Alice!
}
```

- 람다식에서도 로컬 리턴을 사용할 수 있다.
- 람다식안에서 로컬 리턴은 break와 비슷한 역할
- 로컬 리턴은 람다의 실행을 끝내고 람다를 호출했던 코드의 실행을 계속 이어간다.
    - 비로컬 리턴과 구분하기 위해 레이블을 사용해야한다.

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

fun lookForAlice(people: List<Person>) {
    people.forEach {
        if (it.name != "Alice") return@forEach
        print("Found Alice!")
    }
}
```

- 람다를 인자로 받는 인라인 함수의 이름을 return 뒤에 레이블로 사용해도 된다.
- 레이블을 명시하면 함수 이름을 레이블로 사용할 수 없다.

- 람다식에서는 레이블이 2개 이상 붙을 수 없다.
- 비로컬 리턴은 장황하고, 람다 안에 리턴식이 여럿 들어가는 경우는 사용이 불편하다.
    - 다른해벌: 익명 함수

익명 함수: 기본적으로 로컬 리턴

```
data class Person(val name: String, val age: Int)

val people = listOf(Person("Alice", 29), Person("Bob", 31))

fun lookForAlice(people: List<Person>) {
    people.forEach(fun(person) { // 종료 지점
        if (person.name == "Alice") return // 리턴은 가장 가까운 함수를 가리키는데, 이
        println("${person.name} is not Alice")
    })
}

fun main() {
    lookForAlice(people)
    // Bob is not Alice.
}
```

- 익명 함수는 다른 함수에 전달할 수 있는 코드 블록을 작성하는 다른 방법이다.
    - 이름이 없는 일반 함수처럼 생긴 함수로 fun 키워드를 사용하고 함수의 이름을 생략해 정의한다
    - 파라미터 타입도 컴파일러가 추론할 수 있어 생략한다
- 하지만 람다와 익명함수는 리턴 식을 쓸 수 있다는 점에서 차이가 있다.
- 익명 함수는 빠른 리턴이 많이 들어 있어서 람다 구문으로 쓸 때 레비을을 많이 붙여야 하는 코드를 짧게 쓸 때 도움이 된다.

요약

- 함수 타입을 사용해 함수에 대한 참조를 담는 변수나 파라미터나 반환값을 만들 수 있다.
- 고차 함수는 다른 함수를 인자로 받거나 함수를 반환한다.
    - 함수의 파라미터 타입이나 반환 타입으로 함수 타입을 사용하면 고차 함수를 선언할 수 있다.
- 인라인 함수를 컴파일할 때 컴파일러는 그 함수의 본문과 그 함수에게 전달된 람다의 본문을 컴파일한 바이트코드를 모든 함수 호출 지점에 직접 넣어 준다.
    - 이렇게 만들어지는 바이트코드는 람다를 활용한 인라인 함수 코드를 풀어서 직접 쓴 경우와 비교할 때 아무 부가 비용이 들지 않는다.
- 고차 함수를 사용하면 컴포넌트를 이루는 각 부분의 코드를 더 잘 재사용할 수 있다.
    - 또한 고차 함수를 활용해 강력한 제네릭 범용 라이브러리를 만들 수 있다.
- 인라인 함수에서는 람다 안에 있느 리턴문이 바깥쪽 함수를 반환시키는 비로컬 리턴을 사용할 수 있다.
- 익명함수는 람다식을 대신할 수 있으며 리턴식을 처리하는 규칙이 일반 람다식과는 다르다.
    - 반환 지점이 여럿 있는 코드 블록을 만들어야 한다면 람다 대신 익명 함수를 쓸 수 있다.