

# Chapter#8 고차함수

## Features

- 함수 타입
- 고차 함수와 코드를 구조화할 때 고차 함수를 사용하는 방법
- 인라인 함수
- non-local return과 label
- 무명 함수

## 고차함수란,

- 다른 함수를 인자로 받거나 함수를 반환하는 함수
  - i. 람다나 함수 참조를 인자로 전달 받거나
  - ii. 람다나 함수 참조를 반환 하거나



우리는 이미 고차함수를 알.고.있.다

다음의 코드를 보자.

```
val list = //...  
list.filter { x > 0 }
```

우리는 이미 고차함수를 알고.있.다

```
inline fun <T> Array<out T>.filter(  
    predicate: (T) -> Boolean  
): List<T>
```

## 함수 타입

- 람다를 인자로 받는 함수를 정의하려면, 람다 인자의 타입을 어떻게 선언할 수 있는지 알아야 한다.

## 함수 타입

- 고차함수로 전달하려는 람다에 대하여 타입을 선언하는 연습이 필요하다.

```
val sum = { x: Int, y: Int -> x + y }  
val action = { println("class 코틀린 : Kotlin") }
```

코틀린의 타입 추론으로 인해 `sum` 과 `action` 의 변수 타입은 함수 타입

## 함수 타입

- 함수 타입을 정의하는 방법의 핵심 키워드는 3가지 이다.
  - 해당 함수의 매개변수를 명시 하는 `()`
  - 함수의 매개변수와 반환 타입을 구분 해주는 `->`
  - 함수의 반환 타입 `T`

## 함수 타입

- `sum` 과 `action` 변수를 이용하여 함수의 타입 선언에 대한 연습.

```
// val sum = { x: Int, y: Int -> x + y }  
val sum: (Int, Int) -> Int = { x, y -> x + y }  
  
// val action = { println("class 코틀린 : Kotlin") }  
val action: () -> Unit = { println("class 코틀린 : Kotlin") }
```

단, 일반 함수와 다르게 주의해야 할 점은 `Unit` 키워드



## 함수 타입

- $(\text{Int}, \text{Int}) \rightarrow \text{Int}$
- $(\text{Int}, \text{Int}) \rightarrow \text{Int}?$
- $((\text{Int}, \text{Int}) \rightarrow \text{Int})?$

함수 타입이 어떻게 선언되냐에 따라 전혀 다른 반환 타입을 갖는 함수 타입이 된다.

## 함수 타입 #Tip

- 함수 타입에서 매개변수의 이름을 미리 지정할 수도 있다.

```
fun performRequest(  
    url: String,  
    callback: (code: Int, content: String) -> Unit  
) {  
    /*...*/  
}
```

```
val url = "https://api.github.com"  
performRequest(url) { code, content -> /*...*/ }
```

## 인자로 받은 함수 호출

- 지금까지는 고차함수를 구현하기 위한 사전 준비
- 지금부터 고차함수를 어떻게 구현하는지 알아보자.

김슬기 그루

아이, X발 새X야

## 인자로 받은 함수 호출

- 간단한 고차 함수 정의하기

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

고차 함수의 정의 중, 1. 람다나 함수 참조를 인자로 전달 받거나 에 해당

## 인자로 받은 함수 호출

- 코틀린 표준 라이브러리인 `filter` 재구현 해보기

```
// 표준 라이브러리의 filter 구현  
// inline fun <T> Array<out T>.filter(  
//     predicate: (T) -> Boolean  
// ): List<T>
```

```
fun String.filter(  
    predicate: (Char) -> Boolean  
): String
```

## 인자로 받은 함수 호출

- 코틀린 표준 라이브러리인 `filter` 재구현 해보기

```
fun String.filter(predicate: (Char) -> Boolean): String {  
    val sb = StringBuilder()  
    for (index in 0 until length) {  
        val element = get(index)  
        if (predicate(element)) sb.append(element)  
    }  
  
    return sb.toString()  
}
```

```
println("ab1c".filter { it in 'a'..'z' })
```

## 자바에서 코틀린 함수 타입 사용

- 컴파일된 코드에서의 함수 타입은 일반 인터페이스로 바뀐다.
- 즉, FunctionN 인터페이스를 구현하는 객체에 저장한다.
  - 매개변수 개수에 따라 Function0<R>, Function1<P1, R>... 인터페이스 제공
  - 각 인터페이스에 들어 있는 `invoke` 메서드를 호출하여 실행

## 자바에서 코틀린 함수 타입 사용

```
fun processTheAnswer(f: (Int) -> Int) {  
    /*...*/  
}
```

```
// Java8  
// Java8에 추가된 람다를 이용하면 자동으로 함수 타입의 값으로 변환 된다.  
processTheAnswer(number -> number + 1)  
  
// Java7  
// Function 인터페이스의 invoke 메소드를 구현하여 익명 클래스를 넘기면 된다.  
processTheAnswer(new Function<Integer, Integer>() {  
    @Override  
    public Integer invoke(Integer number) {  
        /*...*/  
    }  
})
```



**디폴트 값을 지정한 함수 타입 파라미터나 널이 될 수 있는 함수 타입 파라미터**

- 음... 도대체가 뭘 소리신지...

## 디폴트 값을 지정한 함수 타입 파라미터나 넓이 될 수 있는 함수 타입 파라미터

- 쉽게 말해, 고차 함수의 인자로 전달 될 람다에 디폴트 값 지정이 가능하단 소리
- 3장에서 살펴 봤던 `joinToString` 함수로 사례를 알아보자.

```
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)  
        result.append(element) // 기본 toString 을 이용함.  
    }  
  
    result.append(postfix)  
  
    return result.toString()  
}
```

## 디폴트 값을 지정한 함수 타입 파라미터나 널이 될 수 있는 함수 타입 파라미터

```
val corps = ["Apple", "Google", "Samsung", "LG", "Facebook"]  
println(corps.joinToString())  
  
>>> Apple, Google, Samsung, LG, Facebook
```

- 만약, 출력되는 내용을 모두 소문자 혹은 대문자로 바꾸고 싶다면 (?)

## 디폴트 값을 지정한 함수 타입 파라미터나 널이 될 수 있는 함수 타입 파라미터

- 개발자의 두뇌가 회전하기 시작합니다...
  - i. 기본으로 실행되는 `toString` 메서드 대신 실행 할 수 있는 방법을 생각한다
  - ii. 코틀린에서 함수는 일급 객체 라는 사실을 깨달음
  - iii. `jsonToString` 함수의 매개변수로 함수를 넘겨주자! (람다 또는 함수타입)
- 문제점
  - i. 특수한 케이스로 인해 고차함수로 만든 `jsonToString` 을 호출할 때 마다 람다를 인자로 전달하여야 하므로 기존에 편하게 호출하던 부분도 불편해짐

## 디폴트 값을 지정한 함수 타입 파라미터나 넓이 될 수 있는 함수 타입 파라미터

- 함수 인자에 디폴트 값을 지정 하듯, 디폴트 람다 실행문을 지정

```
fun <T> Collection<T>.joinToString(  
    separator: String = ", ",  
    prefix: String = "",  
    postfix: String = "",  
    transform: (T) -> String = { it.toString() }  
): String {  
    val result = StringBuilder(prefix)  
  
    for ((index, element) in this.withIndex()) {  
        if (index > 0) result.append(separator)  
        result.append(transform(element)) // 전달된 람다 인자를 이용  
    }  
  
    result.append(postfix)  
  
    return result.toString()  
}
```

```
println(corps.jsonToString() { it.toLowerCase() })
```

## 함수를 함수에서 반환

고차 함수의 정의 중, 2. 람다나 함수 참조를 반환 하거나 에 해당

- 함수의 반환 타입으로 함수 타입을 지정해야 한다.
- 함수를 반환하려면 `return` 식에 람다나 멤버 참조 또는 함수 타입의 값을 계산하는 함수를 넣으면 된다.

## 함수를 함수에서 반환

- Example# 1 - 사용자가 선택한 배송 수단에 따른 배송비 계산

```
enum class Delivery { STANDARD, EXPEDITED }

data 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 }
}
```

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

## 함수를 함수에서 반환

- Example# 2 - GUI 연락처 관리 앱 (?)

```
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) {  
            return startsWithPrefix  
        }  
  
        return { startsWithPrefix(it) && it.phoneNumber != null }  
    }  
}
```

```
val contacts = listOf(Person("Dmitry", "Jemerov", "123-4567"), Person("Svetlana", "Isakova", nu  
val contactListFilters = ContactListFilters()  
with (ContactListFilters) {  
    prefix = "Dm"  
    onlyWithPhoneNumber = true  
}  
  
println(contacts.filter(contactListFilters.getPredicate()))
```



## 고차함수란,

- 다른 함수를 인자로 받거나 함수를 반환하는 함수
  - i. 람다나 함수 참조를 인자로 전달 받거나
  - ii. 람다나 함수 참조를 반환 하거나
- 코드 구조를 개선하고 중복을 없앨 때 쓸 수 있는 아주 강력한 도구이다.

## 람다를 활용한 중복 제거

- 함수 타입과 람다 식은 재활용하기 좋은 코드를 만들 때 쓸 수 있는 훌륭한 도구다.
- 람다를 사용할 수 없는 환경에서는 아주 복잡한 구조를 만들어야만 피할 수 있는 코드 중복도 람다를 활용하면 간결하고 쉽게 제거할 수 있다.

## 람다를 활용한 중복 제거 - 사이트 방문 기록 분석 사례

```
enum class OS { WINDOWS, LINUX, MAC, IOS, ANDROID }

data class SiteVisit(
    val path: String,
    val duration: Double,
    val os: OS
)

val log = listOf(
    SiteVisit("/", 34.0, OS.WINDOWS),
    SiteVisit("/", 22.0, OS.MAC),
    SiteVisit("/login", 12.0, OS.WINDOWS),
    SiteVisit("/signup", 8.0, OS.IOS),
    SiteVisit("/", 16.3, OS.ANDROID)
)
```

## 람다를 활용한 중복 제거 - 사이트 방문 기록 분석 사례

#. 특정 OS의 평균 방문 시간을 계산하자.

```
val averageWindowsDuration = log
    .filter { it.os == OS.WINDOWS }
    .map(SiteVisit::duration)
    .average()
```

```
println(averageWindowsDuration)
```

```
>>> 23.0
```

## 람다를 활용한 중복 제거 - 사이트 방문 기록 분석 사례

#. OS 정보를 인자로 전달받는 함수를 작성하여 중복코드를 제거하자.

```
fun List<SiteVisit>.averageDurationFor(os: OS) =  
    filter { it.os == os }.map(SiteVisit::duration).average()
```

```
println(log.averageDurationFor(OS.WINDOWS))  
>>> 23.0
```

```
println(log.averageDurationFor(OS.MAC))  
>>> 22.0
```

하지만, 만약 모바일 디바이스 사용자의 평균 시간을 구해야 된다면, 결국 다시 아래와 같이 하드코딩 되어야 한다.

```
val averageMobileDuration = log  
    .filter { it.os in setOf(OS.IOS, OS.ANDROID) }  
    .map(SiteVisit::duration)  
    .average()  
  
println(averageMobileDuration)  
>>> 12.15
```

즉, 위에서 정의한 확장 함수는 무쓸모가 되었다... 😞

## 람다를 활용한 중복 제거 - 사이트 방문 기록 분석 사례

#. 고차함수를 사용해 중복을 제거하자.

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

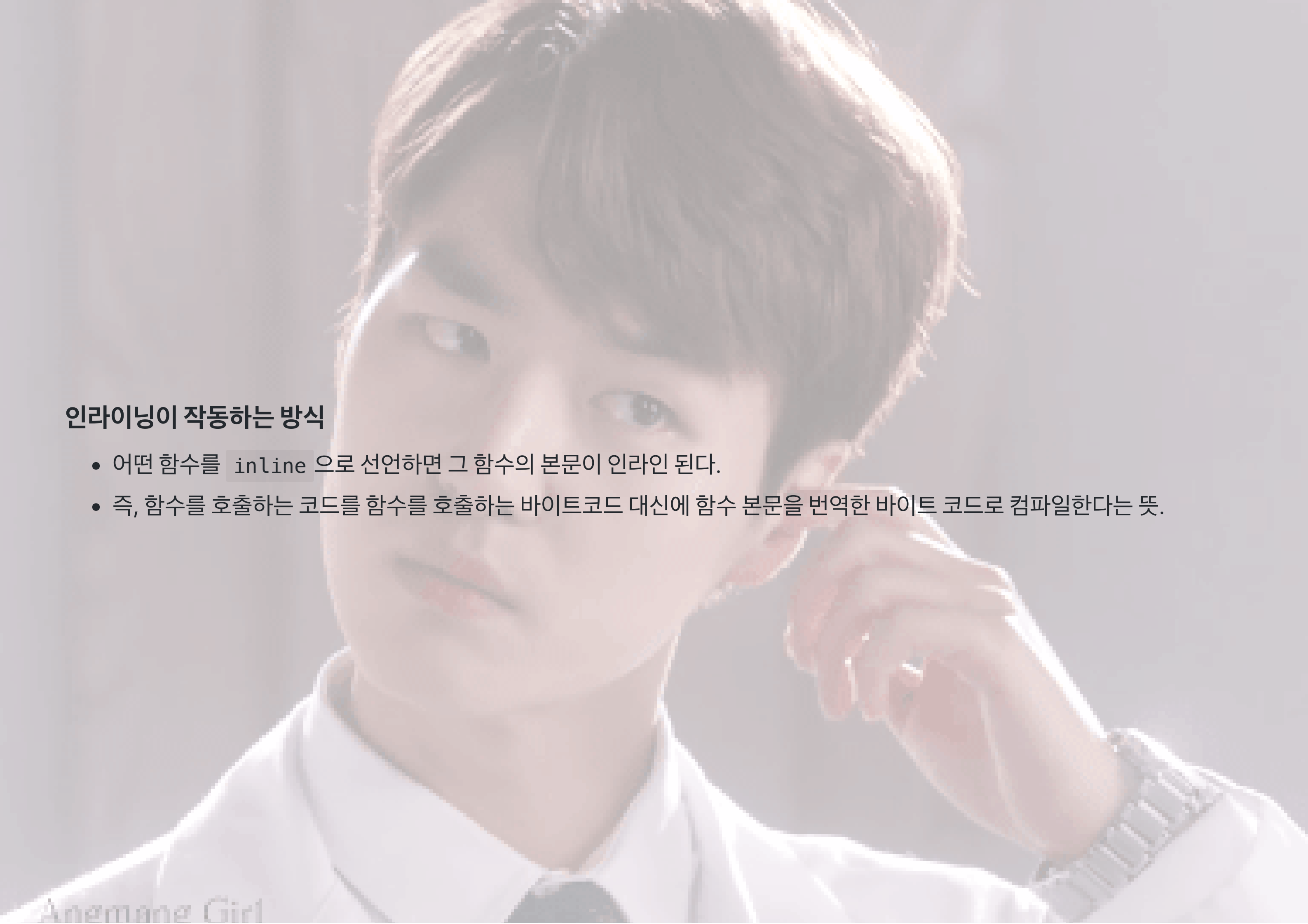
```
println(log.averageDurationFor { it.os in setOf(OS.ANDROID, OS.IOS) })  
>>> 12.15
```

```
println(log.averageDurationFor {  
    it.os == OS.IOS && it.path == "/signup"  
})  
>>> 8.0
```

`filter` 에서 실행되어야 할 함수를 함수 타입 으로 받아서 처리함으로 인해서,  
중복 코드의 제거와 더불어 더욱 더 다양한 조건으로 결과를 추출할 수 있는 확장함수가 되었다. 🙊

## 인라인 함수: 람다의 부가 비용 없애기

- 람다가 변수를 포획하면 생성 시점마다 새로운 익명 클래스 객체가 생기므로, 실행 시점에 익명 클래스 생성에 따른 부가 비용이 있다. (5장)
- 즉, 람다를 사용하는 구현은 똑같은 작업을 수행하는 일반 함수를 사용한 구현보다 덜 효율적이다. (🐒!?)
- 하지만, 우리의 코틀린 컴파일러는 `inline` 변경자를 통해 함수를 호출하는 모든 문장을 함수 본문에 해당하는 바이트코드로 바꿔치기 해줌으로 인해 효율적인 코드를 생성하게 만들어 준다.



## 인라이닝이 작동하는 방식

- 어떤 함수를 `inline` 으로 선언하면 그 함수의 본문이 인라인 된다.
- 즉, 함수를 호출하는 코드를 함수를 호출하는 바이트코드 대신에 함수 본문을 번역한 바이트 코드로 컴파일한다는 뜻.



## 인라이닝이 작동하는 방식: 코드로 알아보기

- 다중 스레드 환경에서 동시 접근을 막기 위한 코드 예시

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

```
fun foo(l: Lock) {  
    println("Before sync")  
  
    synchronized(l) {  
        println("Action")  
    }  
  
    println("After sync")  
}
```

## 인라이닝이 작동하는 방식: 코드로 알아보기

```
fun foo(l: Lock) {  
    println("Before sync")  
  
    lock.lock()  
  
    try {  
        println("Action")  
    } finally {  
        lock.unlock()  
    }  
  
    println("After sync")  
}
```

- `synchronized` 함수의 본문뿐 아니라 `synchronized` 전달된 람다의 본문도 함께 인라이닝 된다는 것에 유의.

## 인라인 함수의 한계

- 인라이닝을 하는 방식으로 인해 람다를 사용하는 모든 함수를 인라이닝할 수는 없다.
  - | 인라인 함수의 본문에 람다가 직접 펼쳐지기 때문에 함수가 매개변수로 받은 람다를 본문에 사용하는 방식으로 한정됨.
- 즉, 매개변수로 받은 람다를 다른 변수에 저장하고 나중에 그 변수를 사용하는 방식이라면, 람다를 표현하는 객체가 어딘가에 존재해야 하므로 람다를 인라이닝 할 수 없다.
  - | `Illegal useage of inline-parameter` 라는 메시지와 함께 인라이닝을 금지시킨다.

## 인라인 함수의 한계

- 둘 이상의 람다를 인자로 받는 함수에서 일부 람다만 인라이닝 하고 싶은 경우 `noinline` 변경자를 이용하면 된다.

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit { /*...*/ }
```

9.2.4절에서 `noinline` 을 사용해야 하는 상황을 몇가지 볼 수 있으므로 8장에서는 여기까지만 언급...

## 컬렉션 연산 인라이닝

- 코틀린 표준 라이브러리의 컬렉션 함수는 대부분 람다를 인자로 받는다.
- 표준 라이브러리 함수를 사용하지 않고 직접 연산을 구현한다면?

```
data class Person(val name: String, val age: Int)
val people = listOf(Person("Alice", 29), Person("Bob", 31))
```

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

```
val result = mutableListOf<Person>()
for (person in people) {
    if (person.age < 30) result.add(person)
}

println(result)
>>> [Person(name="Alice", age=29)]
```

코틀린의 `filter` 함수는 인라인 함수이다.

즉, `filter` 함수를 호출하는 위치에 `filter` 함수의 본문으로 바꿔치기 되므로, 성능상 차이가 없다.

## 함수를 인라인으로 선언해야 하는 경우

- `inline` 키워드의 이점을 배우고 나면 코드를 더 빠르게 만들기 위해 코드 여기저기에 `inline` 을 사용하고 싶어질 것이다.
  - 하지만 좋은 생각은 아니다.
- `inline` 키워드를 사용해도 람다를 인자로 받는 함수만 성능이 좋아질 가능성이 높다.

## 람다를 인자로 받는 함수를 인라이닝하면 생기는 이익

- 인라이닝을 통해 없앨 수 있는 부가 비용이 상당하다.
  - 함수 호출 비용을 줄일 뿐 아니라, 람다를 표현하는 클래스와 람다 인스턴스에 해당하는 객체를 만들 필요가 없어짐.
- 현재의 JVM은 함수 호출과 람다를 인라이닝해 줄 정도로 똑똑하지 못하다.
- 일반 람다에서는 사용할 수 없는 몇가지 기능을 사용할 수 있다.
  - `non-local return`

하지만 `inline` 키워드를 함수에 붙일 때는 코드 크기에 주의를 기울여야 함.

인라이닝하는 함수가 큰 경우 함수의 본문에 해당하는 바이트코드를 모든 호출 지점에 복사하기 때문에, 전체적인 바이트코드가 아주 커질 수 있다.

## 고차 함수 안에서 흐름 제어

- Loop와 같은 명령형 코드를 람다로 바꾸기 시작하면 return 문제에 부딪침.
- 인자로 전달받는 람다 안에서 return을 사용하면 어떤 일이 벌어질지 몇가지 예제를 살펴보자.

## 람다 안의 return 문: 람다를 둘러싼 함수로부터의 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")
}

lookForAlice(people)
>>> Found!
```

위 예제 코드를 `forEach` 로 바꿔도 안절할까?



## 람다 안의 return 문: 람다를 둘러싼 함수로부터의 반환

```
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")
}
```

그렇다. 안전하다.

- 람다 안에서 `return` 을 사용하면 람다로부터만 반환되는 것이 아닌, 람다를 호출하는 함수 자체가 실행을 끝내고 반환 된다.
- 자신을 둘러싸고 있는 블록보다 더 바깥의 다른 블록을 반환하게 만드는 것을 `non-local return` 이라고 부른다.
- 단, `return` 이 바깥쪽 함수를 반환시킬 수 있는 경우는 람다를 인자로 받는 함수가 **인라인 함수**인 경우 뿐이다.
  - 즉, `forEach` 함수는 인라인 함수이며 람다 본문과 함께 인라이닝 된다.
- 반대로, 인라이닝 되지 않는 함수에 전달되는 람다 안에서는 `return` 을 사용할 수 없다.
  - 인라이닝 되지 않는 함수는 람다를 변수에 저장할 수도 있고,
  - 바깥 함수로부터 반환된 뒤에 저장해 둔 람다가 호출될 수도 있다.
  - 즉, 람다 안의 `return`이 실행되는 시점이 바깥쪽 함수를 반환시키기에 너무 늦은 시점일 수 있다.

## 람다로부터 반환: 레이블을 사용한 return

- 람다 식에서도 `local return` 을 사용할 수 있다.
- 람다 안에서 `local return` 은 `for loop`의 `break` 와 비슷한 역할을 한다.
- `local return` 과 `non-local return` 을 구분하기 위해서는 `label` 을 사용해야 한다.
- `return` 으로 실행을 끝내고 싶은 람다 식 앞에 `label` 을 붙이고, `return` 키워드 뒤에 `label` 을 추가하면 된다.

## 람다로부터 반환: 레이블을 사용한 return

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

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

    println("Alice might be somewhere")
}

lookForAlice(people)
>>> Alice might be somewhere
```

- `return` 으로 실행을 끝내고 싶은 람다 식 앞에 `label` 을 붙이고: `people.forEach labelName@{`
- `return` 키워드 뒤에 `label` 을 추가하면 된다.: `return@labelName`
- `local return` 은 for loop의 `break` 와 비슷한 역할을 한다.: `println("Alice might be somewhere")`

람다에 레이블을 붙여서 사용하는 대신 인라인 함수의 이름을 `return` 뒤에 레이블로 사용해도 된다.

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

## 무명 함수: 기본적으로 `local return`

- 무명 함수는 일반 함수와 비슷해 보이지만, 차이점은 **함수 이름**이나 **매개변수 타입**을 생략할 수 있다.
- 몇가지 무명 함수에 대한 예제를 살펴본다.

## 무명 함수

### Example# 1. 무명 함수 안에서 return 사용하기

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

fun lookForAlice(people: List<Person>) {
    people.forEach(fun (person) {
        if (person.name == "Alice") return

        println("${person.name} is not Alice")
    })
}

lookForAlice(people)
>>> Bob is not Alice
```

### Example# 2. filter에 무명 함수 넘기기

```
people.filter(fun (person): Boolean {
    return person.age < 30
})
```

### Example# 3. 식이 본문인 무명 함수 사용하기

```
people.filter(fun (person) = person.age < 30)
```

## 무명 함수: 정리

- 무명 함수 안에서 `label` 이 붙지 않은 `return` 은 무명 함수 자체를 반환시킨다. (**local return**)
- 람다식은 `fun` 키워드를 사용해 정의되지 않으므로, 람다 본문의 `return` 은 람다 밖의 함수를 반환 시키지만, (**non-local**)
- 무명 함수는 `fun` 키워드를 사용해 정의되므로 그 함수 자신이 바로 가장 안쪽에 있는 `fun` 으로 정의된 함수이다.
- 즉, 무명 함수 본문의 `return` 은 그 무명 함수를 반환시키고 (**local return**) 무명 함수 밖의 다른 함수를 반환 (**non-local return**) 시키지 못한다.