8장 고차함수

신림프로그래머 최범균

- 함수 타입
- 고차 함수와 코드 구조화에 고차 함수 사용
- 인라인 함수
- 비로컬 리턴과 레이블
- 무명 함수

함수 타입

```
(Int, String) -> Unit
(x: Int, y: String) -> Unit // 파라미터에 이름 지정 가능, 코드 가독성 위함
```

- ->의 좌측: 파라미터 타입
- ->의 우측: 리턴 타입

예:

```
// 함수 타입 명시
val sum: (Int, Int) -> Int = { x, y -> x + y }
val action: () -> Unit = { println(42) }

// 타입 추론
val sum = { x:Int , y: Int -> x + y }
```

함수 호출

일반 함수 호출과 같은 구문

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

fun main(args: Array<String>) {
    twoAndThree( {a, b -> a + b } )
}
```

자바에서 코틀린 함수 호출

- 코틀린 함수는 FunctionN 인터페이스 타입 구현 객체(자바의 SAM 타입)

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

```
processTheAnswer(number -> number + 1);
```

- 코틀린 함수 타입의 invoke 메서드 호출 가능

자바에서 코틀린 함수 호출

- 확장 함수 호출시 첫 번째 인자로 수신 객체 전달
- Unit 리턴 타입은 Unit.INSTANCE 명시적으로 변환

```
// Collection의 forEach 확장 함수를 자바에서 호출
List<String> strings = ...;
CollectionsKt.forEach(strings, s -> { //
    System.out.println(s);
    return Unit.INSTANCE;
});
```

함수 타입 파라미터의 디폴트 값

- 함수 타입 파라미터 뒤에 람다를 넣으면 됨

```
fun <T> Collection<T>.joinToSTring(
    separator: String = ", ",
    prefix: String = "",
    postfix: String = "",
    transform: (T) -> String = { it.toString() }
): String {
    val result = StringBuilder(prefix)

    this.forEachIndexed { index, element ->
        if (index > 0) result.append(separator)
        result.append(transform(element))
    }
    result.append(postfix)
    return result.toString()
}
```

널 가능 함수 타입 호출

널 안전 연산자(?.)로 invoke 메서드 호출

```
fun <T> Collection<T>.joinToSTring(
    separator: String = ", ",
    prefix: String = "",
    postfix: String = "",
    transform: ((T) -> String)? = null
): String {
    val result = StringBuilder(prefix)

    this.forEachIndexed { index, element ->
        if (index > 0) result.append(separator)
        result.append(transform?.invoke(element) ?: element.toString())
    }
    result.append(postfix)
    return result.toString()
}
```

함수에서 함수 리턴

- 리턴 타입으로 함수 타입 사용

```
fun getShippingCostCalculator(delivery: Delivery): (Order) -> Double {
    if (delivery == Delivery.EXPEDITED) {
        return { order -> 6 + 2.1 * order.itemCount }
    }
    return { order -> 1.2 * order.itemCount }
}

val cal = getShippingCostCalculator(Delivery.EXPEDITED)
println(cal(someOrder))
```

고차 함수를 이용한 중복 제거

- 다른 부분을 함수로 받아서 처리(전략 패턴)

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

List<SiteVisit> log = ...
log.averageDurationFor { it.os in setOf(OS.ANDROID, OS.IOS) }
log.averageDurationFor { is.os == OS.IOS && it.path == "/signup" }
```

인라인 함수

- inline으로 선언한 함수
  - 그 함수의 본문이 호출 위치에 인라인
  - 호출시 전달한 람다도 인라인

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

인라인 함수 호출 변환 예

```
val lock = ReentrantLock()

val result: Int = synchronized(lock) {
    println("some log")
    calculate()
}
println(result)
```

```
ReentrantLock lock = new ReentrantLock();

((Lock)lock).lock();

int var8;
try {
  String var3 = "some log";
  System.out.println(var3);
  var8 = calculate();
} finally {
  ((Lock)lock).unlock();
}
System.out.println(var8);
```

인라인 한계

- 함수 본문에서 파라미터로 받은 람다를 호출하면 쉽게 호출을 람다 본문으로 변경 가능
- 람다른 다른 변수에 저장하고 나중에 그 변수를 사용하면, 람다를 인라인할 수 없음

인라인 함수에서 함수 타입 파라미터를 다른 변수에 저장하면 컴파일 에러

```
inline fun <T> synchronized(lock: Lock, action: () -> T): T {
    ...
    val someA = action // -> 컴파일 에러
    ...
}

Illegal usage of inline-parameter 'action' ...
```

인라인 제외

함수 파라미터를 인라인에서 제외하려면 noinline 수식어 붙임

```
inline fun foo(f: () -> Unit, noinline g: () -> Unit) {

}
```

자바에서 코틀린에 정의한 인라인 함수를 호출해도 인라인하지 않음

컬렉션 연산 인라이닝

- 컬렉션의 filter나 map 등의 함수도 인라인 함수
  - 인라인되므로 중간 리스트 생성

```
people.filter { it.age > 30 } // 중간 결과를 담는 콜렉션 생성
      .map(Person::name)
```

- 시퀀스는 중간 콜렉션 생성이 없으므로 부가 비용 없음
- 콜렉션 크기가 작은 경우는 시퀀스보다 콜렉션 연산이 더 성능이 나을 수 있음

언제 인라인으로 선언하나

- 일반 함수 호출은 JVM이 이미 강력하게 인라인 지원
  - JIT에 발생하므로, 바이트 코드 수준에서는 중복 없음
- 람다를 인자로 받는 함수는 인라인하면 이익이 더 많음
  - 람다 호출 비용 감소
  - 람다를 위한 객체 생성 감소
  - 현재 JVM은 함수 호출과 람다를 인라인할 만큼 똑똑하지 못함
- 인라인은 바이트코드 크기를 증가시키므로 inline 함수 크기는 작아야 함

인라인 함수 사용 예: use

- use는 Closeable에 대한 확장 함수로 자바의 try-with-resource와 같은 기능 제공

```
fun readFirstLineFromFile(path: String): String {
    BufferedReader(FileReader(path)).use { br ->
        return br.readLine()
    }
}
```

람다 안의 return: 넌로컬 리턴

- 람다를 둘러썬 함수로부터 리턴
  - 인라인 함수에 전달된 람다의 경우 넌로컬 리턴 가능
  - 인라인되지 않는 함수에 전달된 람다에서는 return 사용할 수 없음

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

람다에서 레이블을 사용한 리턴

```
fun lookForAlice(people: List<Person>) {
    people.forEach label@ { // @으로 레이블 지정
        if (it.name == "Alice") return@label // 레이블 사용해서 람다에서 리턴
    }
    println("Alice might be somewhere")
}
```

```
fun lookForAlice(people: List<Person>) {
    people.forEach {
        if (it.name == "Alice") return@forEach // 함수이름을 레이블로 사용
    }
    println("Alice might be somewhere")
}
```

무명함수는 로컬 리턴

```
fun lookForAlice(people: List<Person>) {
    people.forEach(fun (person) {
        if (it.name == "Alice") return // 가장 가까운 무명 함수에서 리턴
        println("${person.name} is not Alice")
    })
}
```

- return은 가장 가까운 fun 키워드를 사용한 함수에서 리턴