5장 람다로 프로그래밍

신림프로그래머 최범균

- 람다 식과 멤버 참조
- 함수형 스타일 콜렉션 조작
- 시퀀스
- 자바 함수형 인터페이스를 코틀린에서 사용
- 수신 객체 지정 람다

람다식

함수 타입에 넘길 수 있는 작은 코드 조각

코틀린 람다 식은

- 항상 중괄호로 둘러싸여 있다
- 인자 목록 주변에 괄호가 없다
- 화살표(->)가 인자 목록과 람다 본문을 구분

```
{ x: Int, y: Int -> x + y }
```

람다식 사용

```
val sum = { x: Int, y: Int -> x + y } // 람다 변수 할당
sum(1, 2) // sum은 함수 타입. 함수 실행

{ println(42) }() // 람다 만들자마자 바로 실행
run { println(42) } // run 함수로 람다 실행

val people = listOf(Person("Alice", 20), Person("Bob", 31))
people.maxBy( { p: Person -> p.age } )
people.maxBy { p: Person -> p.age } // 람다가 마지막 인자면, 괄호 밖에 위치 가능

people.maxBy { p -> p.age } // 파라미터 추론 가능하면 생략 가능
people.maxBy { it.age } // 인자가 한 개고 타입 추론 가능하면 디폴트 이름인 it 사용 가능

val sum = { x: Int, y: Int ->
    println("$x와 $y의 합 계산")
    x + Y // 람다 본문이 여러줄이면 마지막 식이 람다의 결과
}
```

람다에서 파라미터나 변수 접근

```
fun printMessagesWithPrefix(
        messages: Collection<String>,
        prefix: String) {
    messages.forEach {
        // 람다에서 바깥 함수의 파라미터 접근
        println("$prefix $it")
    }
}
```

```
fun printProblemCounts(response: Collection<String>) {
    var clientErrors = 0
    var serverErrors = 0
    response.forEach {
        if (it.startsWith("4")) {
            // 람다 밖의 변수 접근/수정 가능
            clientErrors++
        } else {

        }
    }
}
```

멤버 참조

이미 선언된 함수를 값으로 사용해야 할 때 멤버 참조 사용

- 멤버 참조는 프로퍼티나 메서드를 단 하나만 호출하는 함수 값을 생성
- ::는 클래스 이름과 참조하려는 멤버 이름 사이에 위치
- 멤버 참조는 그 멤버를 호출하는 람다와 같은 타입
  - Person::age = person -> person.age
- 멤버 참조 뒤에는 괄호를 넣으면 안 됨

```
val getAge = Person::age // '::'를 이용한 멤버 참조, 확장 함수도 동일
getAge(somePerson)

run(::salute) // 최상위 함수 참조

// Person::age(person) : 안 됨

fun sendMail(p: Person, msg: String): Unit = ...
val nextAction = ::sendMail // (p: Person, msg: String): Unit 타입 함수

val createPerson = ::Person // 생성자 참조
```

바운드 멤버 참조 (1.1부터)

```
// 1.1부터 가능
val p = Person("Dmitry", 34)
val ageFunc = p::age // p에 엮인 멤버 참조
println(ageFunc())
```

콜렉션 API

```
list.filter { it % 2 == 0} // 조건을 충족하는 것만 걸러냄
list.map { it * it } // 값을 변환해서 새로운 콜렉션 생성
list.count // Int
list.all { it > 3 } // boolean, 모든 원소가 만족하면 true
list.any { it > 2 } // boolean, 하나라도 만족하면 true
list.find { it > 3 } // 널가능 타입, 조건을 충족하는 첫 번째 원소 (firstOrNull과 동일)

map.filter { entry -> entry.key % 2 == 0 } // entry: Map.Entry
map.map { entry -> entry.key * 2 to entry.value }
// 맵은 filterKeys, mapKeys, filterValues, mapValues 제공
```

groupBy

```
val strs = listOf("12", "345", "11", "456")
val grouped: Map<Int, List<String>> = strs.groupBy { it.length }

// grouped = {2=[12, 11], 3=[345, 456]}
```

flatMap

펼침 함수

```
val books = listOf(
        Book("동용", listOf("작가1", "작가2")),
        Book("시집", listOf("작가3", "작가1")))

val authors = books
        .flatMap { it.authros } // List<String> [작가1, 작가2, 작가3, 작가1]
        .toSet()
```

콜렉션의 연산자는 즉시 생성함

```
list.filter { it % 2 == 0 } // 새로운 콜렉션 생성
    .map { it * 2 } // 새로운 콜렉션 생성
```

원소 개수가 많고 변환 연산이 많으면 주의 필요

지연 계산 컬렉션: 시퀀스

자바 8의 스트림과 동일하게 최종 연산에 대해서만 결과 생성

```
people.asSequence() // 시퀀스로 변환
      .map(Person::name) // 중간 연산
      .filter { it.startsWith("A") } // 컬렉션과 같은 API 제공
      .toList() // 다시 콜렉션으로 변환
```

자바 8의 스트림과 유사하므로, 자바 8을 지원하지 않는 환경에서 시퀀스 사용

시퀀스 만들기

generateSequence() 함수를 사용해서 시퀀스 생성

```
val numbers = generateSequence(0) { it + 1 }
// 첫 번째 인자 : 초기값
// 두 번째 인자 : 다음 값 생성 로직

fun File.isInsideHiddenDirectory() =
    generateSequence(this) { it.parentFile }.any { it.isHidden }
```

자바 메서드에 람다 전달

함수형 인터페이스를 인자로 취하는 자바 메서드를 호출할 때 람다를 넘길 수 있음

```
// 자바
void postponeComputation(int delay, Runnable computation)
```

```
postponeComputation(10) { println(42) } // 함수형 인터페이스에 람다 전달
postponeComputation(10, object: Runnable { // 객체 식도 가능
    override fun run() {
        println(42)
    }
})

// SAM 생성자: 람다를 함수형 인터페이스의 인스턴스로 변환
val comp: Runnable = Runnable { println(42) }
```

- 컴파일러가 무명 클래스와 인스턴스를 만들어줌

with

- with: 첫 번째 인자로 받은 객체를 두 번재 인자로 받은 람다의 수신 객체로 만든다

```
fun alphabet(): String {
    val sb = StringBuffer()
    return with(sb) {
        ('A'..'Z').forEach { ch -> this.append(ch) }
        append("\nNow I know tthe alphabet!") // this를 생략해도 sb.append 호출
        this.toString() // this는 with의 첫 번째 인자로 전달한 sb
    }
}

// <T, R> with(receiver: T, block: T.() -> R) // block 함수의 수신 객체는 T
```

apply

- apply: 자신에게 전달된 객체를 반환한다는 점 빼면 with와 거의 같다.

```
fun alphabet(): String = StringBuilder().apply {
    ('A'..'Z').forEach { ch -> append(ch) }
    append("\nNow I know tthe alphabet!")
}.toString()

// fun <T> T.apply(block: T.() -> Unit): T
```

- apply는 확장 함수