# 4부

## 1. 람다

```
fun main() {
    val list = listOf(1,2,3,4)
    val result = list.map({n: Int -> "[$n]" })
    result eq listOf("[1]","[2]","[3]","[4]")

    // list 는 List<Int> 로 람다가 n 이 Int 라는 걸 타입 추론 할 수 있다. 즉 빼도 됨.
    val result2 = list.map({n -> "[$n]" })
    // 파라미터가 하나일 경우 코틀린은 자동으로 파라미터 이름을 it 로 만든다.
    // 즉 개발자가 변수 지정하지 않아도 된다. 더 깔끔
    val result3 = list.map({"[$it]"})
    // 함수의 파라미터가 람다뿐이면 람다 주변 괄호를 없을 수 있다. 더 깔끔
    val result4 = list.map{"[$it]"}
    // 함수가 여러 파라미터를 받고 마지막 파라미터가 람다인 경우에는 인자 목록을 감싼 괄호 다
    val result5 = list.joinToString(" ") { "[$it]" } eq [1] [2] [3] [4]
    // 이름 붙인 안자로 호출 가능하다. 이때는 람다를 괄호 안에 넣어야 한다.
    val result6 = list.joinToString(separator = " ", transform = { "[$it]" })

    val listc = listOf('a','b','c')
    // list mapIndexed 라이브러리
    listc.mapIndexed { index, element -> "[$index: $element]" } eq
        listOf("[0: a]", "[1: b]", "[2: c]")
    // 특정 인자를 사용하지 않는 경우 _ 를 사용하면 컴파일 경고 무시 가능
    listc.mapIndexed { index, _-> "[$index]" } eq listOf("[0]", "[1]", "[2]")
    // list indices 라이브러리
    listc.indices.map { "[$it]" } eq listOf("[0]","[1]","[2]")

}
```

- 함수 리터럴이라고 부르기도 하며, 익명 함수로 부터 만들어졌다.
  함수 생성에 필요한 최소한의 코드만 필요하고, 다른 코드에 람다를 직접 삽입할 수 있다.
- 람다의 파라미터가 없는 경우에는 화살표를 사용하지말자 (코틀린 스타일 가이드)

## 2. 람다의 중요성

```
fun main() {
    val list = listOf(1,2,3,4)
    val isEven = { e: Int -> e % 2 == 0 }
```

```
    list.filter(isEvent) eq listOf(2,4)

    // 클로저
    val list2 = listOf(1,5,7,10)
    var sum = 0
    val divider = 5
    list.filter { it % divider == 0 }
        .forEach { sum += it }
    sum eq 15
}
```

- 람다를 var 나 val 에 담을 수 있고, 재 사용할 수 있다.
- 람다의 또 다른 특징으로는 자신의 영역 밖에 있는 요소를 참조할 수 있는 능력이 있다. (클로저 - 다른 언어(js 등)에 클로저가 의미하는 것과 다름)

### 3. 컬렉션에 대한 (람다) 연산

```
fun main() {
    // List 생성자에는 인자가 두개이다. MutableList 도 마찬가지
    // 첫 번재는 크기이고 두 번째는 List 의 각 원소를 초기화하는 람다
    val list1 = List(10) {it}
    list1 eq "[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]"
    val list2 = List(10) {0}
    list1 eq "[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]"
    val list3 = List(10) {'a' + it}
    list1 eq "[a, b, c, d, e, f, g, h, i, j]"

}
```

- 미리 작성된 람다 `(Supplier<T>, Consumer<T>, Function<T, R>, Predicate<T>)`
  - filter
  - map
  - forEach
  - any
  - all
  - none
  - find
  - lastOrNull
  - count
  - partition (predicate (true, false) 로 Pair 객체를 만든다.)
  - sumBy, sumByDouble

- sorted, sortedBy, sortedDescending, sortedByDescending
- take 첫 번째 원소를 취함
- drop 첫 번째 원소를 제거
- 그 밖에도 코틀린에서는
  - filterNot 반대 원소를 가져올 수 도 있다.
  - filterNotNull null 이 아닌 원소를 반환해줌

## 4. 멤버 참조

- 함수, 프로퍼티, 생성자에 대해 만들수 있는 멤버 참조는 해당 함수, 프로퍼티, 생성자를 호출하는 뻔한 람다를 대신할 수 있다.
- 멤버 참조

```
data class Message(
    val sender: String,
    val text: String,
    val isRead: Boolean
)

fun main() {
    val messages = listOf(
        Message("kitty", "h", true),
        Message("kitty", "h2", false),
        Message("bose", "meet today", false)
    )
    // Message::isRead 를 멤버 참조라고 한다.
    val unread = messages.filterNot(Message::isRead)
    unread.size eq 1

    message.sortedWith(compareBy(
        // sort 우선순위 1. 읽지 않은 순서 2. 메시지 보낸 사람 순서
        Message::isRead, Message::sender)) eq
        listOf(
            Message("bose", "meet today", false)
            Message("kitty", "h2", false),
            Message("kitty", "h", true),
}
```

- 함수 참조

```
data class Message(
    val sender: String,
    val text: String,
    val isRead: Boolean,
    val attachments: List<Attachment>
```

```
)

data class AttachMent(
    val type: String,
    val name: String
)

fun Message.isImportant(): Boolean =
    text.contain("[긴급]") ||
    attachment.any {
        it.type == "image" &&
            it.name.contain("상용")
        }

fun main() {
    val messages = listOf(
        Message("Boss", "Let's discuss goals for next year", false,
            listOf(Attachment("image", "상용 긴급")))
    // 함수 참조
    messages.any(Message::isImportant) eq true
}
```

  - 만약 최상위 수준 함수에 대해서는 클래스 이름이 없음으로 `::function` 처럼 쓴다.
  - 확장 함수에 경우는 `확장된 클래스::function` 로 쓴다.

- 생성자 참조

```
data class Student(
    val id: Int,
    val name: String
)

fun main() {
    val names = listOf("Alice", "Bob")
    val students = names.mapIndexed { index, name -> Student(index, name)
    students eq listOf(Student(0, "Alice"), Student(1, "Bob"))
    // 생성자 참조
    names.mapIndex(::Student) eq students
}
```

## 5. 고차 함수

- 함수를 다른 함수의 인자로 넘길 수 있거나, 함수가 반환값으로 함수를 돌려주는 것을 의미

```
val isPlus: (Int) -> Boolean = { it > 0 }

// any 구현체
fun <T> List<T>.any(
```

```
    predicate: (T) -> Boolean
): Boolean {
    for (element int this) {
        if (predicate(element))
            return true
    }
    return false
}

fun main() {
    listOf(1, 2, -3).any(isPlus) eq true
}
```

- `(Int) -> Boolean` 은 함수 타입이다. 함수 타입은 0 개 이상의 파라미터 타입 목록을 둘러싼 괄호로 시작하며, 화살표가 따라오고 화살표 뒤에 반환 타입이 온다.
  - `(파라미터타입1, … , 파라미터타입N) -> 반환타입`
  - 반환타입은 null 이 될 수 있는 타입도 가능하다.

## 6. 리스트 조작하기

```
fun main() {
    val left = listOf("a", "b", "c", "d")
    val right = listOf("q", "r", "s", "t")

    // zip 은 두 List 의 원소를 하나씩 짝 짓는 방식인 함수
    left.zip(right) eq "[(a, q), (b, r), (c, s), (d, t)]"
    // 두 시퀀스 중 하나가 원소를 소진하면 묶기 연산도 끝난다.
    left.zip(0..10) eq "[(a, 0), (b, 1), (c, 2), (d, 3)]"

    // zip 2번째 파라미터로 람다를 넣을 수 있다. return Pair
    left.zip(right) {el1, el2 -> Pair(el1, el2)}

    // 한 list 에서 어떤 원소와 그 원소 다음 원소를 묶으려면 zipWithNext() 사용
    left.zipWithNext() eq listOf(
        Pair("a", "b"),
        Pair("b", "c"),
        Pair("c", "d"),
    )

    // flatten 은 list안에 list 인자를 받아서 원소가 따로따로 들어 있는 List 를 반환한다
    val list = listOf(
        listOf(1,2),
        listOf(4,5),
        listOf(6,7)
    )
    list.flatten() eq "[1, 2, 4, 5, 6, 7]"
```

```
    // flatMap 은 java 에서와 동일하게 2중(n) 리스트면 단일(n-1) 리스트로 변경해주는 함수
    val intRange = 1..3
    intRange.flatMap { a ->
        intRange.map { b -> a to b }
    } eq "[" +
        "(1, 1), (1, 2), (1, 3), " +
        "(2, 1), (2, 2), (2, 3), " +
        "(3, 1), (3, 2), (3, 3)" +
        "]"
}
```

## 7. 맵 만들기

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

fun main() {
    val list = listOf(
        Person("박종현", 30),
        Person("박종현", 29),
        Person("김양미", 28)
    )
    // groupBy 의 파라미터는 원본 컬렉션의 원소를 분류하는 기준이 되는 키를 반환하는 람다
    val map: <Map<Int, List<Person>> = list.groupBy(Person::age)
    map[30] eq listOf(Person("박종현", 30))
    map[28] eq listOf(Person("김양미", 28))

    // associateWith 는 원소가 키가되고 value 를 지정된 람다로 반환하는 map 을 만들 수
    val map2: Map<Person, String> = list.associateWith { it.name }
    map2 eq mapOf(
        Person("박종현", 30) to "박종현",
        Person("박종현", 29) to "박종현",
        Person("김양미", 28) to "김양미"
    )

    // associateBy 는 원소가 값이 되고 key 를 지정된 람다로 반환하는 map 을 만들 수 있다
    val map2: Map<String, Person> = list.associateBy { it.name }
    map2 eq mapOf(
        // 키가 유일 하지 않은 경우 실패된다.
        "박종현" to Person("박종현", 30),
        "김양미" to Person("김양미", 28)
    )

    // getOrElse 은 get 시 null 인 경우 인자 값을 return 한다 (default 같은 느낌)
    // getOrPut 은 get 시 null 인 경우, 첫 번재 파라미터의 key 두 번째 값을 계산한 후 p
```

```
    // filterValues value 파라미터
    // filterKeys key 파라미터
    // filter 로는 entry 를 받아서 key, value 를 사용

    // map 을 list 로, map 을 재 구성한 map 으로
    val even = mapOf(2 to "two", 4 to "four")
    even.map { "${it.key}=${it.value}" } eq listOf("2=two", "4=four")
  even.map { -it.key to "minus ${it.value}" }.toMap() eq mapOf(-2 to "minus t
}
```

## 8. 시퀀스

- java 의 Stream 라이브러리로 보면 된다.

```
fun main() {
    val list = listOf(1,2,3,4)
    list.filter{ it % 2 == 0 }
        .map {it * it}
        .any {it < 10 } eq true

    // 위와 같다
    val mid1 = list.filter {it % 2 == 0}
    mid eq listOf(2,4)
    val mid2 = mid1.map {it * it}
    mid2 eq listOf(4, 16)
    mid2.any { it < 10 } eq true

    // 이 녀석이 list 보다 빨리 끝난다.
    val list2 = listOf(1,2,3,4)
    list2.asSequence
        .filter{ it % 2 == 0 }
        .map {it * it}
        .any {it < 10 } eq true
}
```

- List 에 대한 연산은 **즉시 계산**된다. (함수를 호출하자마자 모든 원소에 대해 바로 계산됨)
  - **즉시 계산(수평적 평가)**은 직관적이고 단순하지만 최적은 아니다.
    만약 미리
    `any` 를 만족하는 원소가 있어 연산을 멈출 수 있다면 더 합리적일 것 이다.
- 시퀀스에 대해 **지연 계산**을 수행하는 경우를 **수직적 평가**라고 한다.
  - 지연 계산을 사용하면 어떤 원소와 연관된 값이 진짜 필요할 때만 그 원소와 관련된 연산을 수행한다. 해당 원소가 최종 결과를 찾아내면 나머지 원소는 처리되지 않음.
  - List 에 대해 시퀀스를 적용하려면 `asSequence()` 함수를 사용한다.
  - 시퀀스 연산에는 **중간 연산**과 **최종 연산**이 있다.
    - 중간 연산 - 다른 시퀀스를 return 한다. 일반적 filter, map

        - 최종 연산 - 시퀀스가 아닌 다른 값을 내놓는다. any, sum, count, toList 등
    - 시퀀스는 한 번 만 이터레이션 할 수 있다.  또 시도하면 에러난다.
    - 참고) java Stream 는 지연 계산(수직적 평가)로 된다.
    - `generateSequence()` 를 사용해 무한 시퀀스를 만들 수 있다. 첫 번째 인자는 시퀀스의 첫 번째 원소, 두 번째 인자는 이전 원소로부터 다음 원소를 만들어내는 방법을 정의하는 람다.

## 9. 지역 함수

- Java 와 다르게 함수 안에서 함수를 정의할 수 있다. (ak. 지역 함수)

```
fun main() {
    val logMsg = StringBuilder()
    // 지역함수
    fun log(message: String) = logMsg.appendLine(messge)
    log("starting")
    val x = 42
    log("result: $x")

    logMsg.toString() eq """
        starting
        result: 42
    """
}
```

- 지역 함수는 클로저다. 자신을 둘러싼 환경의 var, val 을 포획한다.
- 람다가 복잡해 읽기 어렵다면, 지역 함수나 익명 함수를 대신해라.
- forEach 는 return 으로 함수를 정지시킬 수 없다. 그러니 forEach 에서 반환하려면 `return@forEach` 이런식으로 해야한다. (이렇게도 가능 `list.forEach tag@` , `return@tag`)
- 지역함수 조작하기

```
// 익명함수
fun first(): (Int) -> Int {
    val func = fun(i: Int) = i + 1
    func(1) eq 2
    return func
}

// 람다
fun second(): (String) -> String {
    val func2 = { s:String -> "$s!" }
    func2("abc") eq "abc!"
    return func2
}

// 지역함수
```

```
fun third(): () -> String {
    fun greet() = "Hi!"
    return ::greet
}

fun fourth() = fun() = "Hi!"

fun fifth() = { "Hi!" }

fun main() {
    val funRef1: (Int) -> Int = first()
    val funRef2: (String) -> String = second()
    val funRef3: () -> String = third()
    val funRef4: () -> String = fourth()
    val funRef5: () -> String = fifth()

    funRef1(42) eq 43
    funRef2("xyz") eq "xyz!"
    funRef3() eq "Hi!"
    funRef4() eq "Hi!"
    funRef5() eq "Hi!"

    first()(42) eq 43
    second()("xyz") eq "xyz!"
    third()() eq "Hi!"
    fourth()() eq "Hi!"
    fifth()() eq "Hi!"
}
```

## 10. 리스트 접기

```
fun main() {
    val list = listOf(1,10,100,1000)
    list.fold(0) { sum, n ->
        sum + n
    } eq 1111

    list.reduce { acc, e ->
        acc + e
    } eq 1111
}
```

- `fold`는 리스트의 모든 원소를 순서대로 조합해 결과값을 하나로 만든다.
  - 첫 번째 인자, 초기값
    두 번째 인자 람다 누적값 (sum), 다음 원소 (n) 정의
- `reduce` 는 초기 값이 없고 최초 원소 값이 초기값이다.

## 11. 재귀

```
// 일반적인 피보나치 수 (이전 값을 사용하지 않아 비효율)
fun fibonacci(n: Long): Long {
    return when(n) {
        0L -> 0
        1L -> 1
        else ->
            fibonacci(n - 1) + fibonacci(n - 2)
    }
}

fun fibonacciWithTailrec(n: Int): Long {
    tailrec fun fibonacciWithTailrec(
        n: Int,
        current: Long,
        next: Long
    ): Long {
        if (n == 0) return current
        return fibonacciWithTailrec(n -1, next, current + next)
    }
    return fibonacciWithTailrec(n, 0L, 1L)
}
```

- 일반적 프로그래밍과 다른게 없음 함수내 자기 함수 호출로 보면 됨.
- 하지만, stackOverFlowError 무한 재귀 되는 경우
  - 이를 방지하고자 함수형 프로그래밍은 **꼬리 재귀(`tailrec`)**를 사용
  - 코틀린에서는 재귀 호출을 반복문으로 변환해 호출 스택 비용을 제거해준다.
    꼬리 재귀는 컴파일러가 수행하는 최적화이지만, 모든 재귀 함수에 적용할 수 있는 건 아니다.
  - `tailrec`을 성공적으로 사용하려면 재귀가 마지막 연산이어야 한다.
    (재귀 함수가 자기 자신을 호출해 얻은 결과값을 아무 연산도 적용하지 않고 즉시 반환해야한다.)