Chapter 03.코틀린 고급 - 11.람다로 프로그래밍하기

람다로 프로그래밍하기

1. 함수형 프로그래밍 정의

- 함수형 프로그래밍은 혹은 FP(Functional-Programming)은 수학의 함수적 개념을 참고해 만든 패러다임의 하나로 깔끔하고 유지보수가 용이한 소프트웨어를 만들기 위해 함수를 사용합니다
- 함수형 패러다임은 부수효과가 없고 똑같은 input이 들어오면 항상 동일한 output을 내놓는 순수함수의 개념을 기반으로 람다, 고차함수, 커리, 메모이제이션, 모나드 등의 개념을 포함한다
- 이러한 함수형 프로그래밍에 대한 자세한 학습은 매우 방대해서 학습의 범위를 벗어나므로 이번 장에선 코틀린에서 어떻게 함수형 패러다임을 지원하는지 기본적인 내용만 학습하겠습니다

---

2. 코틀린과 함수형 프로그래밍

2.1 함수를 값으로 사용하기

- 함수형 프로그래밍에서 함수는 1급 객체로 분류된다
- 1급 객체란 함수를 인자에 넘기거나 변수에 대입하거나 함수를 반환하는 개념 등을 통틀어 1급 객체라고 한다

- 코드를 통해서 1급 객체의 특성을 코틀린에서 어떻게 구현하는지 알아보자
- 함수도 타입이다 `() -> Unit` 화살표 표기법을 사용한다

```
val func: () -> Unit = {}
```

- 함수는 값이 될 수 있고 역으로 값은 함수가 될 수 있다 그러므로 함수에 인자로 넘기거나 데이터 구조에 저장할 수 있다
- 리스트에 저장 후 사용

```
val printHello: () -> Unit = { println("Hello") }

fun main() {
  val list = mutableListOf(printHello)
  list[0]() // 함수 호출을 위해 ()를 써야한다
}
// Hello
```

- 함수는 값이므로 변수로 받아서 처리할 수 있다

```
val list = mutableListOf(printHello)
val func = list[0]
func()

// Hello
```

- 다른 함수의 인자로 사용

```
val printHello: () -> Unit = { println("Hello") }

fun call(block: () -> Unit) {
    block()
}

fun main() {
  call(printHello)
}
// Hello
```

- fun으로 선언된 함수는 값으로 다룰 수 없다

```
fun main() {
  val list = mutableListOf(printNo) // 컴파일 오류
}

fun printNo() {
    println("No!")
}
```

- 인자를 받아 결과를 리턴하는 함수 값 만들어보기

```
val printMessage : (String) -> Unit = { message : String -> println(message) }

// 타입 생략
val printMessage : (String) -> Unit = { message  -> println(message) }

// it 사용
val printMessage : (String) -> Unit = { println(it) }
```

- 다수의 인자를 받아 결과를 리턴하는 함수 값 만들어보기

```
val plus: (Int, Int) -> Int = { a, b -> a + b }

fun main() {
  val result = plus(1, 3)

  println(result)
}
// 4
```

- `고차 함수 (Higher-Order Function, HOF)`라고 한다 고차 함수는 함수를 인자로 받거나 결과로 돌려주는 함수를 의미한다

```
fun main() {

    val list = listOf("a", "b", "c")

    val printStr: (String) -> Unit = { println(it) }
    forEachStr(list, printStr)

}

fun forEachStr(collection: Collection<String>, action: (String) -> Unit) {
    for (item in collection) {
        action(item)
    }
}
// a
// b
// c
```

- 컬렉션의 filter, map, forEach 등도 함수를 인자로 받아서 결과를 반환하므로 고차함수이다

2.2 익명함수와 람다 식

- 함수형 프로그래밍에선 이름 없는 무명 함수를 `익명함수`라고 한다

```
fun outerFunc(): () -> Unit {
    return fun() {
        println("이것은 익명함수!")
    }
}

fun main() {
  outerFunc()()
}
// 이것은 익명함수!
```

- 익명함수를 람다 표현식으로 바꿀 수 있다

```
fun outerFunc(): () -> Unit {
    return { println("이것은 람다함수!") }
}
```

- 조금 더 추상화

```
fun outerFunc(): () -> Unit = { println("이것은 람다함수!") }
```

- 람다 표현식의 전체 구문

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }

// 최대로 생략하면 아래와 같다
val sum2 = { x: Int, y: Int -> x + y }
```

- 사실 앞선 학습에서 람다 표현식을 계속 사용해왔다

```
val printHello: () -> Unit = { println("Hello") }
```

- 함수의 마지막 인자가 함수인 경우 `후행 람다 전달` 을 사용할 수 있다

```
fun main() {

    val list = listOf("a", "b", "c")

    // 제거
    // val printStr: (String) -> Unit = { println(it) }
    forEachStr(list) {
        println(it)
    }

}

fun forEachStr(collection: Collection<String>, action: (String) -> Unit) {
    for (item in collection) {
        action(item)
    }
}
```

- 람다 함수의 파라미터가 1개인 경우에만 `it` 을 사용할 수 있다

```
fun main() {

    arg1 {
        it.length
    }

    arg2 { a, b ->
        a.length
        b.length
    }
}

fun arg1(block: (String) -> Unit) {}

fun arg2(block: (String, String) -> Unit) {}
```

2.3 람다 레퍼런스 사용하기

- 람다 레퍼런스를 사용하면 좀 더 가독성 좋게 함수를 인자로 넘길 수 있다
- 일반적인 함수 값 스타일

```
val callReference: () -> Unit = { printHello() }

callReference()
// Hello
```

- 람다 레퍼런스 사용

```
val callReference = ::printHello
callReference()()
// Hello
```

- numberList를 만들어서 람다 레퍼런스 사용 전

```
val numberList = listOf("1", "2", "3")
numberList.map { it.toInt() }.forEach { println(it) }
```

- 람다 레퍼런스 사용 후

```
numberList.map(String::toInt).forEach(::println)
```

- 탑-레벨 또는 로컬 함수는 `::함수명` 을 사용하고 클래스의 멤버이거나 확장 함수는 클래스 지시자를 사용한다 `클래스명::함수명`

2.4 유용한 함수형 라이브러리 애로우

- 직접 함수를 제작하는 방법도 있지만 고급 기능은 이미 만들어져 있고 검증된 라이브러리를 사용하는 것이 좋다
- 애로우는 코틀린에서 함수형 프로그래밍을 위한 라이브러리이다
- 애로우는 함수형 프로그래밍의 다양한 개념 및 기능들을 제공하고 있고 점점 사용사례가 많아지고 있다
- https://arrow-kt.io/