Chapter 03.코틀린 고급 - 08.페어와 구조분해할당

페어와 구조분해할당

1. 페어

- 먼저 페어에 대해 살펴보기 전에 `Int 타입 인자 2개를 받아서 더하는 함수` 를 만들어 본다

```
// Int 타입 인자 2개를 받아서 더하는 함수
fun plus(a: Int, b: Int) = a + b

fun main() {
    println(plus(1, 3))
}
// 4
```

- 만약 함수형 프로그래밍의 접근법을 사용해서 plus를 수정한다면 `튜플(tuple)` 이라는 개념을 사용할 수 있다
- 수학의 함수 정의를 사용하면 예제와 같다

```
// f((1, 3)) = 1 + 3 = 4
// 이 경우 안쪽의 ()를 생략할 수 있다
// f(1, 3) = 1 + 3 = 4
```

- 함수 정의를 기반으로 코드를 작성하면 튜플이라는 클래스를 만들어 사용한다

```
class Tuple(val a: Int, val b: Int)

fun plus(tuple: Tuple) = tuple.a + tuple.b

fun main(0 {
  println(plus(Tuple(1, 3)))
}
// 4
```

- 이와 같이 튜플을 사용하면 튜플 하나의 인자에 여러 원소를 포함할 수 있다

- 코틀린은 `페어`를 통해 2개의 요소가 있는 튜플을 기본 제공한다

```
fun plus(pair: Pair<Int, Int>) = pair.first + pair.second
```

- 첫번째인자는 `first` 두번째인자는 `second` 로 사용할 수 있다

- 페어는 불변이다

```
val pair = Pair("A", 1)
pair.first = "B" // 컴파일 에러
```

- 내부 코드를 보면 이와 같다

```
public data class Pair<out A, out B>(
    public val first: A,
    public val second: B
) : Serializable {
  //...
}
```

- 내부 코드가 데이터클래스 기반이므로 copy, componentN 함수도 기본 제공한다
- `copy` 사용

```
val newPair = Pair("A", 1).copy(first = "B")
println(newPair)
// (B, 1)
```

- `componentN` 사용

```
val second = newPair.component2()
println(second)
// 1
```

- `toList` 를 사용해 요소를 불변 리스트로 만들 수 있다

```
val list = newPair.toList()
println(list)
// [B, 1]
```

---

2. 트리플

- 페어가 2개의 요소를 가질 수 있다면 트리플은 3개의 요소를 가질 수 있는 튜플이다

```
val triple = Triple("A", "B", "C")
println(triple)
// (A, B, C)
```

- 첫번째인자는 `first` 두번째인자는 `second` 세번째인자는 `third`를 사용할 수 있다

- 페어의 모든 특성을 동일하게 가지고 있다
- 불변성

```
val triple = Triple("A", "B", "C")
triple.third = "D" // 컴파일 에러
```

- `copy` 와 `componentN` 같이 사용

```
val newTriple = triple.copy(third = "D")
println(newTriple.component3())
// D
```

---

3. 구조분해할당

- 구조분해할당을 사용하면 값을 분해해서 한번에 여러 변수를 초기화할 수 있다

```
val (a, b, c) = newTriple
println("$a, $b, $c")
// A, B, D
```

- 이전 예시처럼 타입을 생략할 수 있지만 명시적으로 타입을 선언할 수도 있다

```
val (a: String, b: String, c: String) = newTriple
```

- 구조분해할당은 컴파일러 내부에서 componentN 함수를 사용한다

```
String a = (String)newTriple.component1();
String b = (String)newTriple.component2();
```

```
String c = (String)newTriple.component3();
```

- 배열 또는 리스트에서도 구조분해할당을 사용할 수 있다

```
val list3 = newTriple.toList()
val (a1, a2, a3) = list3
println("$a1, $a2, $a3")
// A, B, D
```

- 즉, 배열 또는 리스트도 componentN이 존재한다는 것이다

```
list3.component1()
```

- 다만 배열이나 리스트는 데이터가 무한정 존재할 수 있으므로 componentN은 앞선 5개 요소에 대해서만 제공한다

```
list3.component1()
list3.component2()
list3.component3()
list3.component4()
list3.component5()
```

- 구조분해할당은 for loop에서도 유용하게 사용된다

```
val map = mutableMapOf("이상훈" to "개발자")
for ( (key, value) in map) {
    println("${key}의 직업은 $value")
}
// 이상훈의 직업은 개발자
```

- 사실 map을 초기화할때 사용하는 `to`는 내부적으로 페어를 사용한다

```
public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

- 따라서 아래와 같이 바꿀 수 있다

```
val map = mutableMapOf(Pair("이상훈","개발자"))
```