객체지향과 클래스 2

코틀린 기초강좌

강사: 배정만

학습요약

- 학습목표: 코틀린에서 사용하는 클래스 학습
- 핵심키워드: Data Classes, Sealed Classes, Enum Classes, Delegation, Properties, Fields
- 학습내용: 코틀린에서 사용하는 데이터 클래스와 같은 프로그래밍 기술에 대해 배웁니다.

데이터 클래스

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

- 컴파일러는 기본 생성자에서 선언 된 모든 속성에서 다음 멤버를 자동으로 파생시킴
  - equals()/hashCode()
  - toString() -> "User(name=John, age=42)'
  - componentN() functions
  - copy()

데이터 클래스 요구 사항

- 기본 생성자에는 하나 이상의 매개 변수가 있어야합니다.
- 모든 기본 생성자 매개 변수는 val 또는 var 로 표시해야합니다.
- 데이터 클래스는 abstract, open, sealed, inner 일 수 없습니다.

멤버 상속 관련 규칙

- 데이터 클래스에서 equals (), hashCode () 또는 toString ()을 명시적으로 구현한 경우 기존 구현이 사용됨
- 수퍼 타입에 open 또는 호환 가능한 타입 componentN() 함수가있는 경우 해당 함수가 데이터 클래스에 대해 생성되고 수퍼 타입 함수를 대체
- componentN () 및 copy () 함수에 대한 명시 적 구현을 제공하는 것은 허용되지 않음

구조 분해
(Destructuring)

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // prints "Jane, 35 years of age”

val name = person.component1()
val age = person.component2()
```

- componentN()는 구조분해 함수를 사용할 수 있게 함
- component1 () 및 component2 () 함수는 Kotlin에서 널리 사용되는 협약(convention)

튜플 (Tuples)

- 튜플(tuple)은 유한 개의 사물의 순서있는 열거이다.
- 임의의 n-튜플은 순서쌍의 개념을 이용하여 재귀적으로 정의된다.
- 5-튜플의 예를 들면 (2, 7, 4, 1, 7)와 같다
- 튜플은 다른 수학 개념들(예를 들어 벡터)을 나타내는 데에 자주 사용된다.

Sealed Class

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()

fun eval(expr: Expr): Double = when(expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
    NotANumber -> Double.NaN
    // the `else` clause is not required because we've covered all the cases
}
```

- enum클래스의 확장
- enum은 단일 인스턴스로만 존재하는 반면, sealed는 서브 클래스를 포함 한 여러 인스턴스를 가질 수 있음
- 클래스는 그 자체로 추상 클래스이며 직접 인스턴스화 할 수 없으며 추상 멤버를 가질 수 있습니다.

Nested 클래스

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

Inner Class

```
class A { // label @A
    inner class B { // label @B
        fun Int.foo() { // label @foo
            val a = this@A // A의 this
            val b = this@B // B의 this
            val c = this // foo()의 Int를 받는다
            val c1 = this@foo // foo()의 Int를 받는다
            val funLit = lambda@ fun String.() {
                val d = this // funLit의 리시버
            }
            val funLit2 = { s: String ->
                val d1 = this // 람다의 명시적 레이블 없으므로 foo()
```

- Inner 클래스는 외부 클래스의 객체에 대한 참조를 전달