Mobile Programming

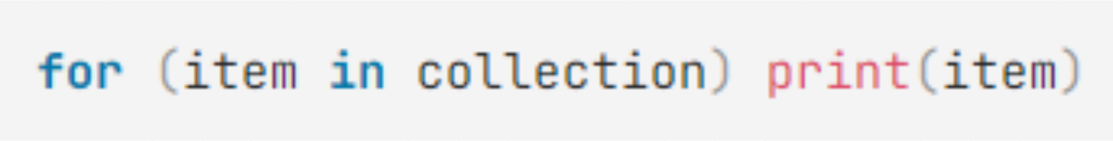
Kotlin Basics – Part II

Agenda

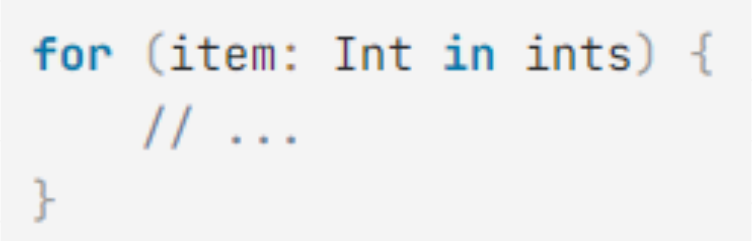
- Loop: for/while
- Function
- OOP

Loop: for (1/2)

- Iterates through anything that provides an iterator
- Basic syntax



```
for (item in collection) print(item)
```



```
for (item: Int in ints) {
    // ...
}
```

instead of collections

- Iterate over a range of numbers ($\boxed{x..y}$) $x \leq y$
  - step : iterate over numbers with an arbitrary step ≥ 1
  - downTo : iterate numbers in reverse order
  - until : iterate a number range which does not include its end element

Loop: for (2/2)

- Example)

```
            (1≤i≤3)
for (i in 1..3) {
    Log.d("ITM","$i")     1 2 3
}
          (6≥i≥0)
for (i in 6 downTo 0) {   6 5 4 3 2 1
    Log.d("ITM","$i")
}
          (1≤ i ≤11)
for (i in 1..11 step 3) {   1 4 7 10
    Log.d("ITM","$i")
}
          (6≥i≥0)
for (i in 6 downTo 0 step 2) {   6 4 2 0
    Log.d("ITM","$i")
}
          (1≤ i <11)
for (i in 1 until 11 step 2){   1 3 5 7 9
    Log.d("ITM","$i")
}
```

```
val arr = IntArray(5){it+1}

for (i in arr) {
    Log.d("ITM","$i")   1 2 3 4 5
}

for ((index, i) in arr.withIndex()) {
    Log.d("ITM","$index's value= $i")
}
```

```
arr.forEach { Log.d("ITM","$it") }   1 2 3 4 5
```

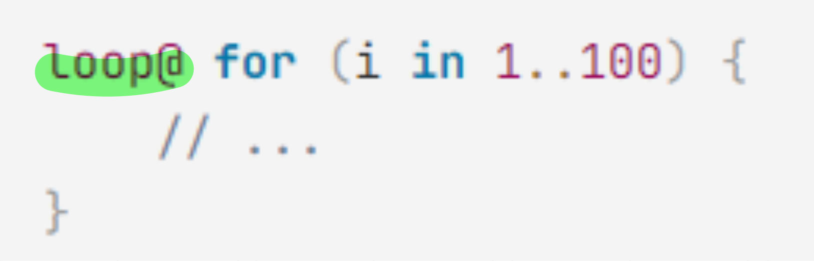
Loop: while

- while
  - Checks the condition first and then executes the body
- do-while
  - Executes the body first and then checks the condition

```
for (i in 1..10) {
    Log.d("ITM", "$i")
}

var num =1
while (num <= 10) {
    Log.d("ITM", "$num")
    num++
}

var num2 =1
do{
    Log.d("ITM", "$num2")
    num2++
} while(num2<=10)
```

Loop: continue & break (1/2)

- continue
  - Proceeds to the next step of the nearest enclosing loop
- break
  - Terminates the nearest enclosing loop
- break/continue with labels
  - Labels have the form of an identifier followed by the @ sign



```
loop@ for (i in 1..100) {
    // ...
}
```

  - break/continue with label breaks/continues the loop specified with that label!

Loop: continue & break (2/2)

- Example)
  - Normal continue/break in the nested loop

```
for (i in 1..3) {
    for (j in 1..5) {
        if (j % 2 == 0) continue // break
        Log.d("ITM", "$i, $j")
    }
}
```

1 1
1 3
1 5

2 1
2 3
2 5

3 1
3 3
3 5

continue

1 1
2 1
3 1

break

  - Labeled continue/break in the nested loop

```
outer@ for (i in 1..3) {
    for (j in 1..5) {
        if (j % 2 == 0) continue@outer // break@outer
        Log.d("ITM", "$i, $j")
    }
}
```

1 1
2 1
3 1

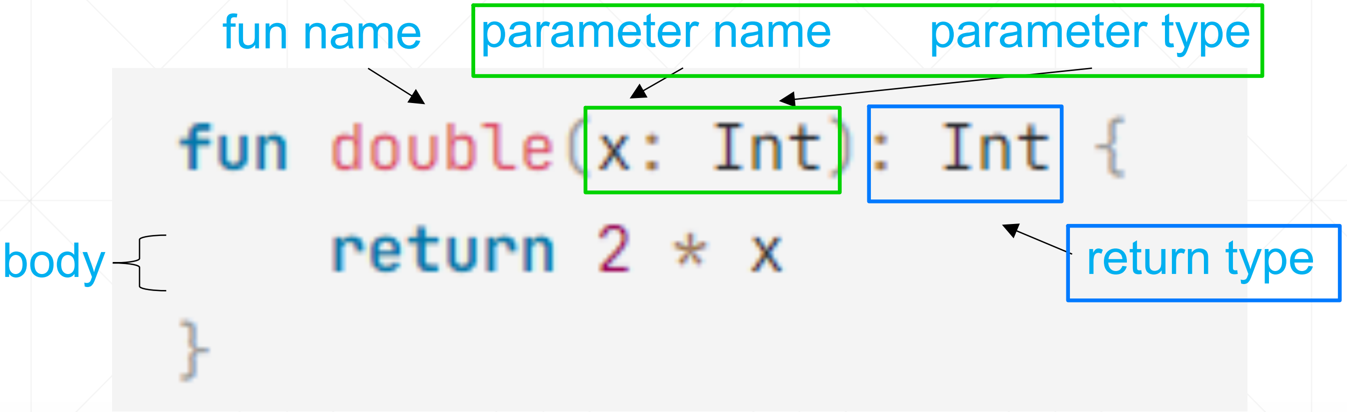
continue

1 1

break

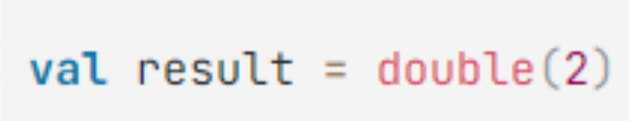
Function (1/8)

- Kotlin functions are declared using the *fun* keyword



```
fun double(x: Int): Int {
    return 2 * x
}
```

- How to use a function?
  - Just call it!



```
val result = double(2)
```

Function (2/8)

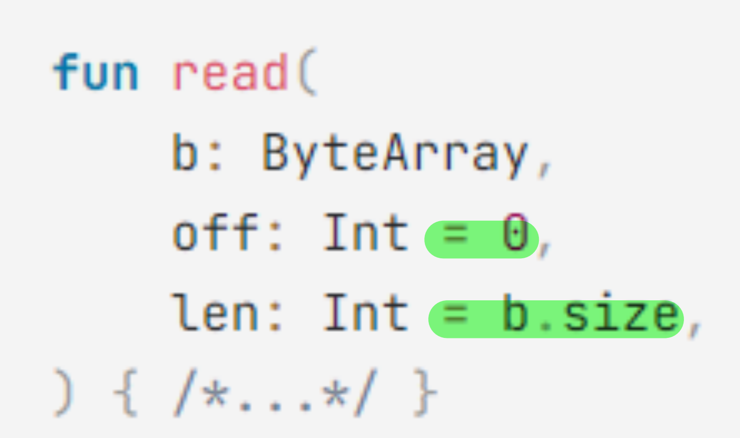
Parameters

- Defined using Pascal notation - name: type
- Separated using commas, and each parameter must be explicitly typed

```
fun powerOf(number: Int, exponent: Int): Int { /*...*/ }
```

Default arguments

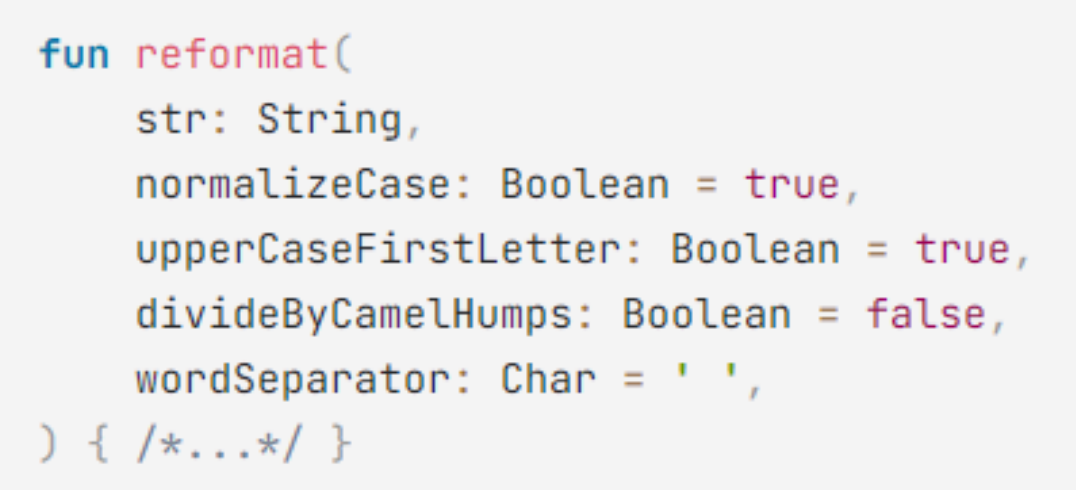
- Parameters can have default values, used when you skip the corresponding argument
- Default value is defined using = after the type



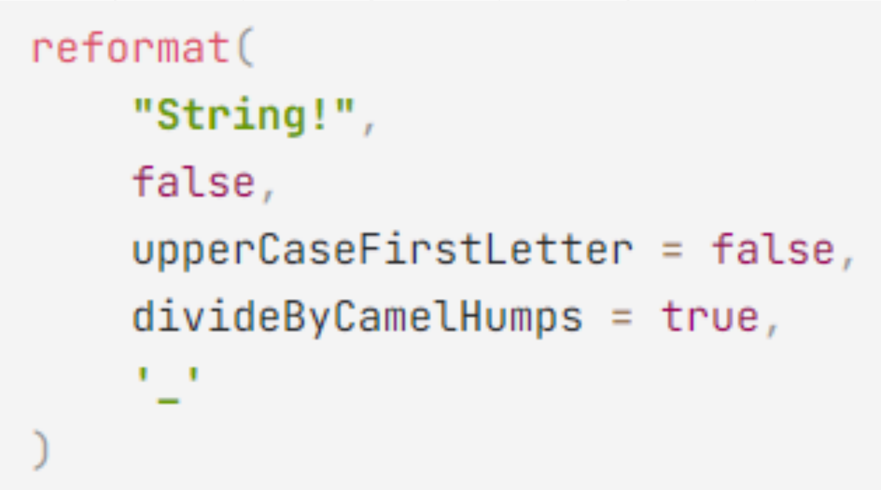
```
fun read(
    b: ByteArray,
    off: Int = 0,
    len: Int = b.size,
) { /*...*/ }
```

Function (3/8)

- Named arguments
  - When calling a function, you can name one or more of its arguments

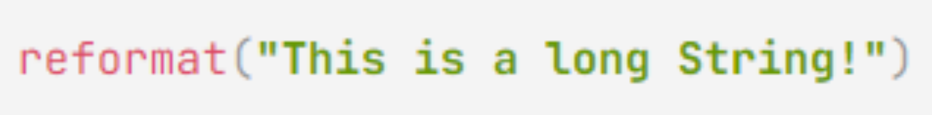


```
fun reformat(
    str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    divideByCamelHumps: Boolean = false,
    wordSeparator: Char = ' ',
) { /*...*/ }
```



```
reformat(
    "String!",
    false,
    upperCaseFirstLetter = false,
    divideByCamelHumps = true,
    '_'
)
```

  - Parameters with default values can be skipped



```
reformat("This is a long String!")
```

Function (4/8)

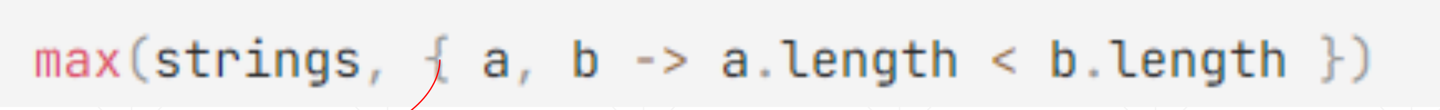
- Unit returning functions
  - Unit: similar to void of Java

- If a function does not return a useful value, then its return type is Unit!
  - This value (Unit) does not have to be returned explicitly
  - The Unit return type declaration is also optional

```
fun printHello(name: String?): Unit {
    if (name != null)
        Log.d("ITM","Hello $name")
    else
        Log.d("ITM","Hi there!")
    // `return Unit` or `return` is optional
}
```

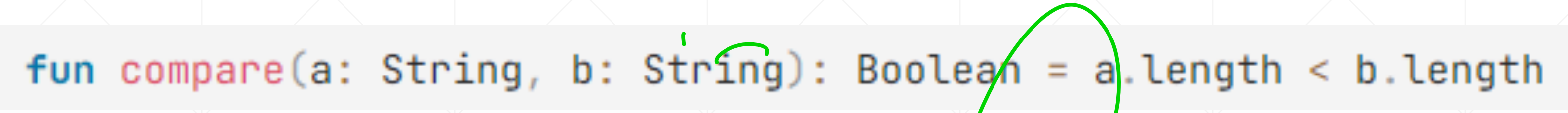
Function (5/8)

- Lambda expression
  - Functions that are not declared but are passed immediately as an expression



```
max(strings, { a, b -> a.length < b.length })
```

Expression that is itself a function



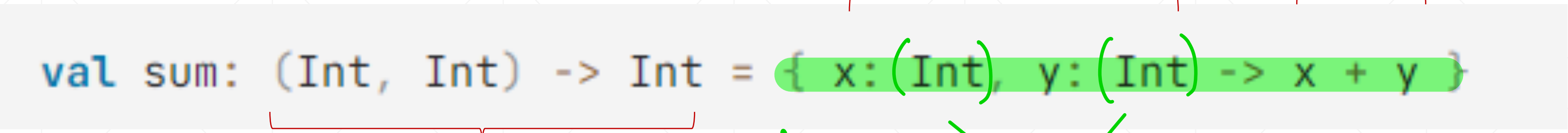
```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

Function (6/8)

- Lambda expression syntax

parameters

body

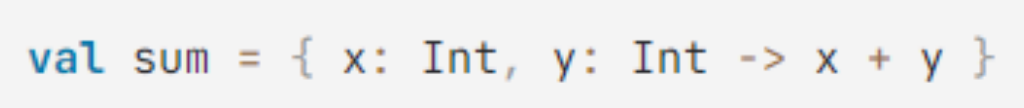


```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

Type annotation ⇒ optional

optional if type annotation exists

- Always surrounded by curly braces
- Parameter declarations in the full syntactic form go inside curly braces and have optional type annotations

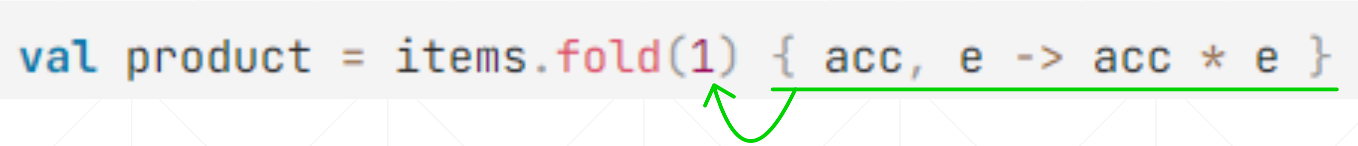


```
val sum = { x: Int, y: Int -> x + y }
```

- The body goes after the `->`
- If the inferred return type of the lambda is not Unit, the last expression inside the lambda body is treated as the return value

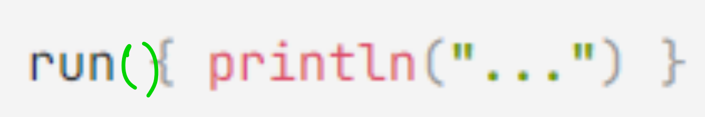
Function (7/8)

- Trailing Lambdas
  - If the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses



```
val product = items.fold(1) { acc, e -> acc * e }
```

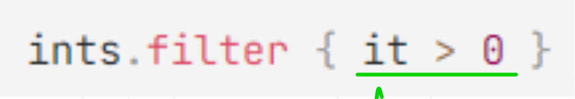
  - If the lambda is the only argument in that call, the parentheses can be omitted entirely



```
run(){ println("...") }
```

Function (8/8)

- it: implicit name of a single parameter
  - If the compiler can parse the signature without any parameters, the parameter does not need to be declared
  - `->` can be omitted
  - The parameter will be implicitly declared under the name *it*



```
ints.filter { it > 0 }
```

↳ λ:Int -> λ > 0

- Returning a value from a lambda expression
  - The value of the last expression is implicitly returned

ints.filter {
  Log.d(tag, msg)
  true
}

Example: Lambda

- Example on Lambda
  - High-order function: a function that takes functions as parameters, or returns a function

```
class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)

      val sum = {x:Int, y:Int -> x+y}
      val multiply:(Int,Int)-> Unit ={ x, y ->
         Log.d("ITM","x * y = ${x * y}")
      }

      Log.d("ITM","by lambda: ${sum(2,3)}")  5
      Log.d("ITM","by function: ${sum1(2,3)}") 5
      multiply(3,4)  12
      Log.d("ITM", highOrderFun({ x:Int, y:Int -> x.toString()+y.toString()})) 27
                                          as parameter
   }

   fun sum1(x:Int, y:Int):Int{
      return x + y
   }

   fun highOrderFun(f:(Int, Int)->(String)): String {
      return f(2,3)
   }

}
```

Example: Lambda

- … from previous lecture

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
Log.d("ITM", numbers.first { it.length > 3 })
Log.d("ITM", numbers.last { it.startsWith("f") })

val numbers2 = listOf(1, 2, 3, 4)
Log.d("ITM","${(numbers2.find { it % 2 == 0 })}")
Log.d("ITM","${numbers2.findLast { it % 2 == 0 }}")
```

- Use of lambda function

```
val numbers2 = listOf(1, 2, 3, 4)
Log.d("ITM", "1: ${numbers2.find({ num:Int -> num % 2 == 0 })}")
Log.d("ITM", "2: ${numbers2.find(){ num:Int -> num % 2 == 0 }}")
Log.d("ITM", "3: ${numbers2.find{ num:Int -> num % 2 == 0 }}")
Log.d("ITM", "4: ${numbers2.find{ num -> num % 2 == 0 }}")
Log.d("ITM", "5: ${numbers2.find{ it % 2 == 0 }}")
```

contextual typing
type inference

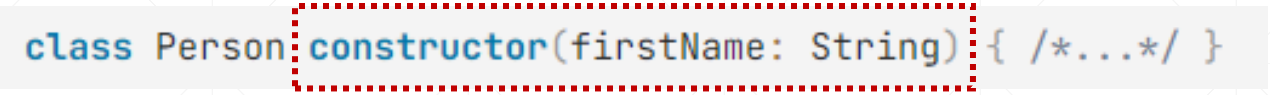
OOP: Class

- Classes in Kotlin are declared using the keyword *class*
  - Class declaration consists of
    - Class name
    - Class header (specifying its type parameters, the primary constructor, and some other things)
    - Class body surrounded by curly braces

```
class ClassName {
        var Variable
        fun Function() {
                // code
        }
}
```

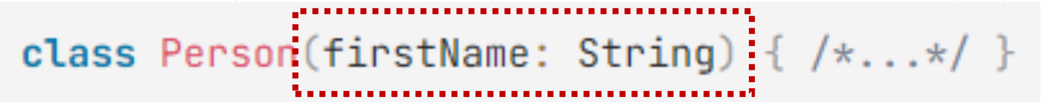
OOP: Constructors (1/5)

- A *primary* constructor
  - Part of the class header



```
class Person constructor(firstName: String) { /*...*/ }
```

  - If the primary constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted



```
class Person(firstName: String) { /*...*/ }
```

  - Initialization code can be placed in initializer blocks prefixed with the *init* keyword

OOP: Constructors (2/5)

- A *primary* constructor
  - The initializer blocks are executed in the same order as they appear in the class body
  - Primary constructor parameters can be used in the initializer blocks as well as property initializers
  - Adding val/var to parameters makes them class properties ⇒ able to use in methods

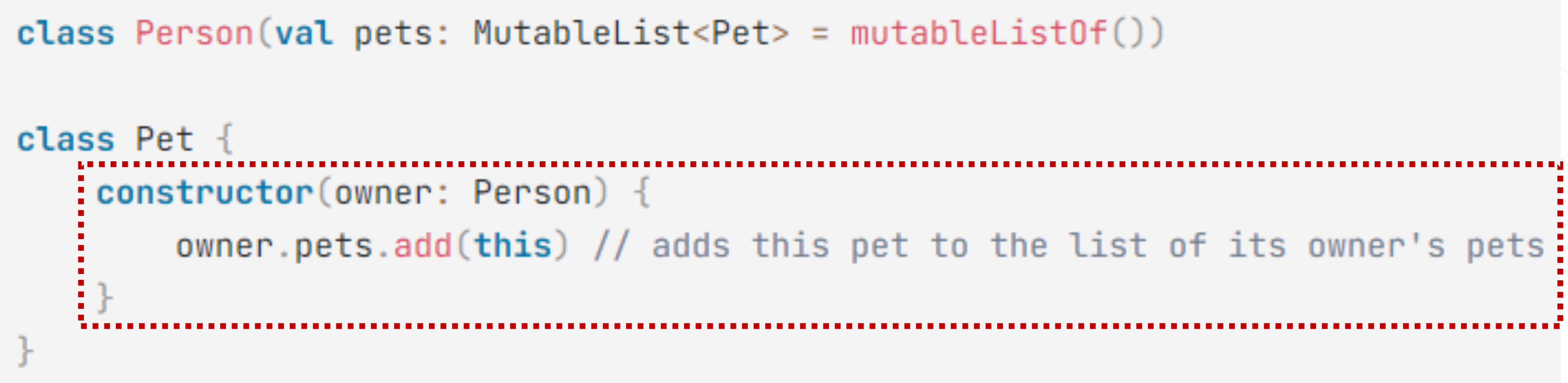
```
                    val/var
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name"

    init {
①       Log.d("ITM","First initializer block that prints ${name}")
    }

    val secondProperty = "Second property: ${name.length}"

    init {
②       Log.d("ITM","Second initializer block that prints ${name.length}")
    }
}
```

Property initializer

Initializer block

OOP: Constructors (3/5)

- One or more *secondary* constructors
  - Prefixed with *constructor*

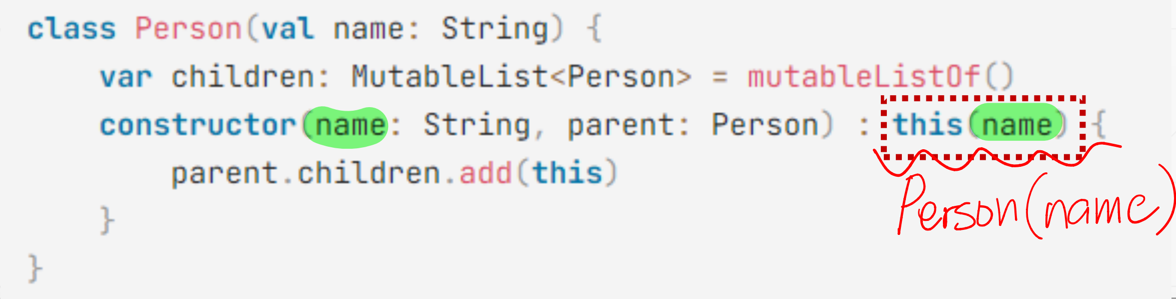


```
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    constructor(owner: Person) {
        owner.pets.add(this) // adds this pet to the list of its owner's pets
    }
}
```

  - If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s)
  - Delegation to another constructor of the same class is done using *this* keyword

val parent = Person("Parent")

val child = Person("Child", parent)



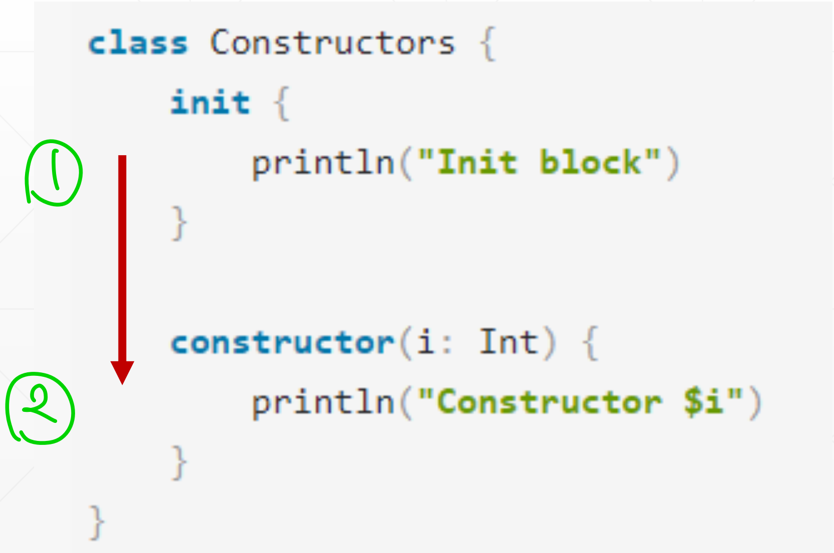
```
class Person(val name: String) {
    var children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Person(name)

OOP: Constructors (4/5)

- Delegation to the primary constructor happens as the first statement of a secondary constructor

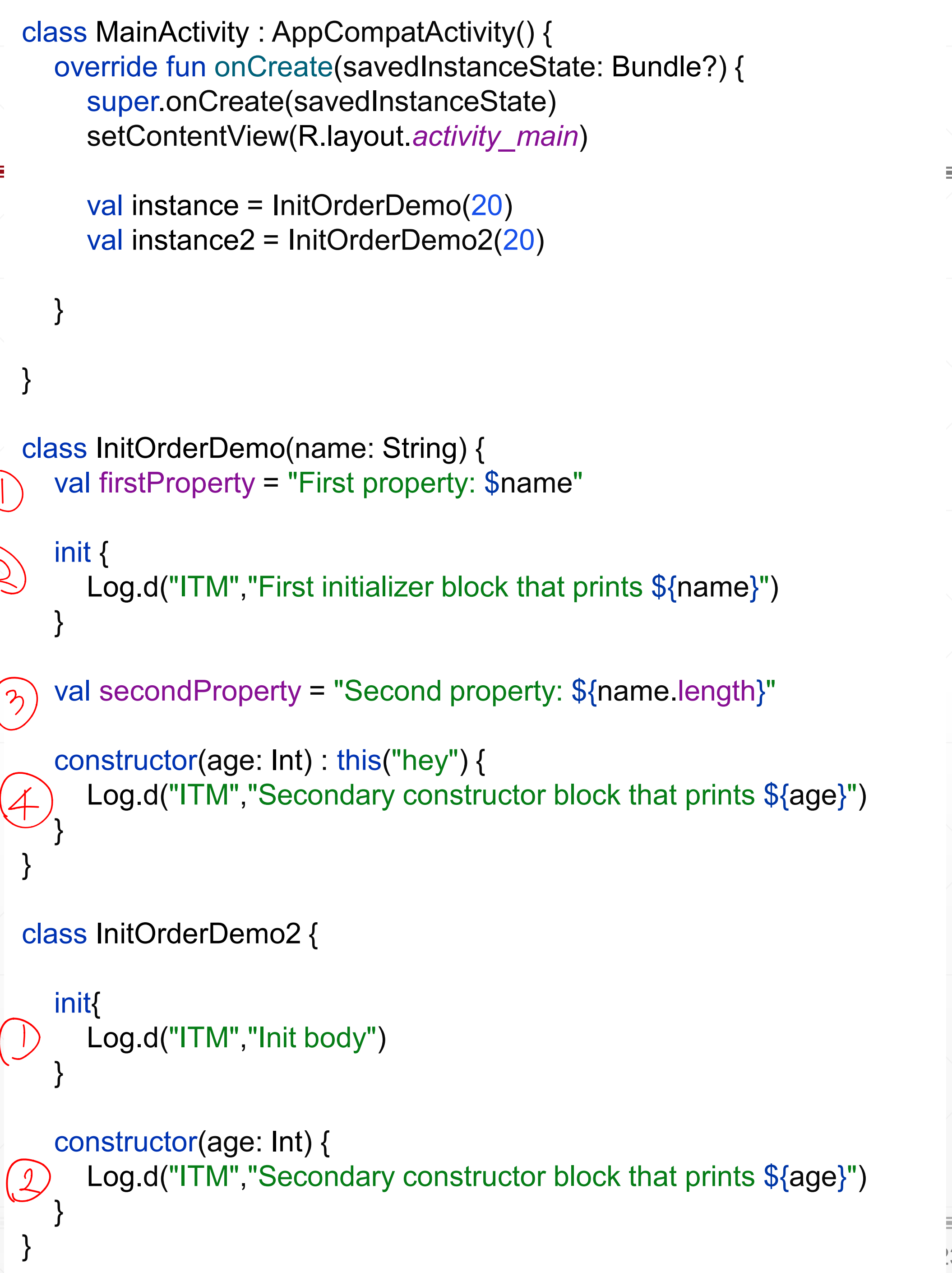
- The code in all initializer blocks and property initializers is executed before the body of the secondary constructor
  - Even if the class has no primary constructor, the delegation still happens implicitly, and the initializer blocks are still executed



```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor $i")
    }
}
```

OOP: Constructors (5/5)

- Example)
  - Class with primary and secondary constructors
  - Class without primary, but with secondary constructor



```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val instance = InitOrderDemo(20)
        val instance2 = InitOrderDemo2(20)

    }

}

class InitOrderDemo(name: String) {
①   val firstProperty = "First property: $name"

②   init {
        Log.d("ITM","First initializer block that prints ${name}")
    }

③   val secondProperty = "Second property: ${name.length}"

    constructor(age: Int) : this("hey") {
④       Log.d("ITM","Secondary constructor block that prints ${age}")
    }
}

class InitOrderDemo2 {

    init{
①       Log.d("ITM","Init body")
    }

    constructor(age: Int) {
②       Log.d("ITM","Secondary constructor block that prints ${age}")
    }
}
```

OOP: Class Properties

- Properties can be declared either as mutable (var), or as read-only (val)

```
class Address {
  var name: String = "default"
     get() = field
     set(value) { field = value+", Korea" }
  val street: String = "Baker"
  val city: String = "London"
  var state: String? = null
  var zip: String = "123456"
}
```

➢ Custom getter/setter is also possible

- Backing field is required if you want to access the property itself

store actual value of property

⇒ access by "field" avoiding recursive calling

class property 실제 저장하는 내부 한.

custom getter, setter 생성 시 field로 구현 가능.

OOP: Object and Companion Object

- Singleton language support
  - You can access the member of Object without instantiation
  - Companion object: Object declaration inside a class Java static
    - marked with the *companion* keyword

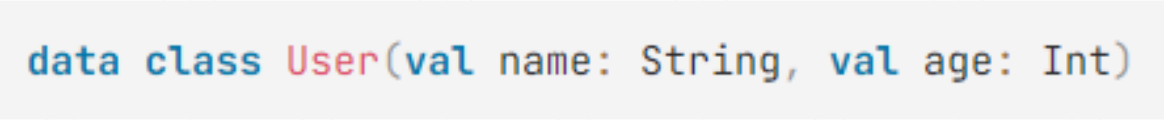
```
object ITM {
    val numStudents = 60
    fun print(){
        Log.d("ITM","we don't love Kotlin")
    }
}

class IE {
    companion object {
        val numStudents = 30
        fun print(){
            Log.d("ITM","we don't like Kotlin")
        }
    }

    fun graduate(){
        Log.d("ITM", "No. Go to graduate school!")
    }
}
```

```
        Log.d("ITM","${ITM.numStudents}") 60
        ITM.print()

//      IE.graduate()
        Log.d("ITM","${IE.numStudents}") 30
        IE.print()

//      val myIE = IE()
//      myIE.graduate()
```

instance required

OOP: Data Class (1/2)

- Classes whose main purpose is to hold data



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

- The compiler automatically derives the following members from all properties declared in the primary constructor:
  - equals()/ hashCode() pair
  - toString() of the form "User(name=John, age=42)“
  - copy()
    - To copy an object, allowing you to alter some of its properties while keeping the rest unchanged
  - …

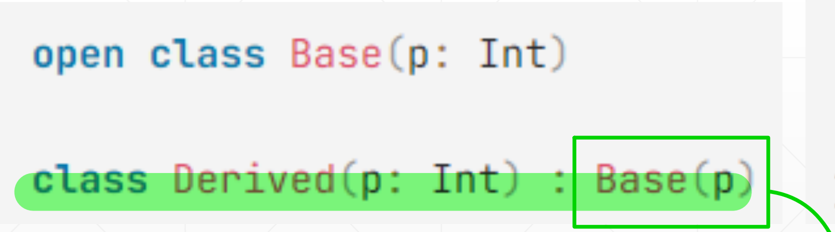
OOP: Data Class (2/2)

- Example)

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

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val uData = User("jinwoo",38)
        Log.d("ITM",uData.toString())   jinwoo, 38
        val uData2 = uData.copy(age=28)
        Log.d("ITM","this is real: ${uData2.toString()}")   jinwoo, 28

    }
}
```

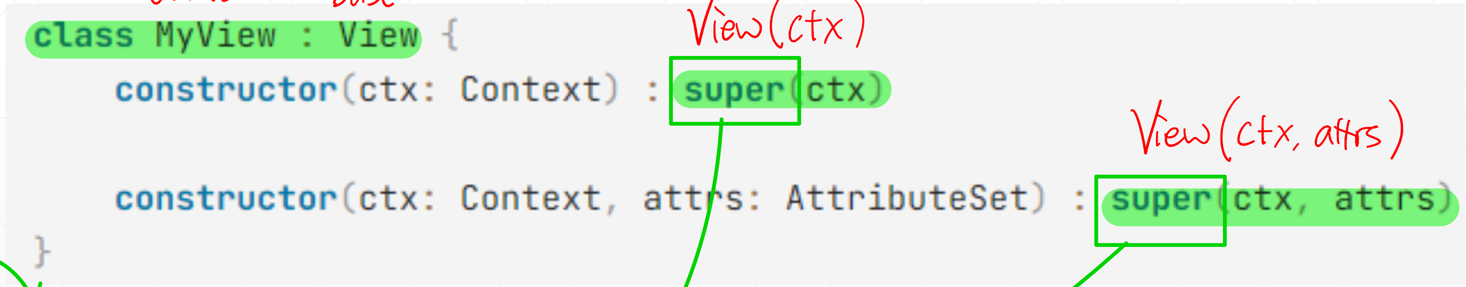
OOP: Inheritance (1/3)

- All classes in Kotlin have a common superclass: *Any*
- By default, Kotlin classes are final – they can't be inherited!
  - To make a class inheritable, mark it with the *open* keyword
- Syntax of inheritance



```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

derived base



```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

View(ctx)

View(ctx, attrs)

  - If the derived class has a primary constructor, the base class must be initialized in that primary constructor according to its parameters
  - If the derived class has no primary constructor, then each secondary constructor has to initialize the base type using the *super* keyword!

OOP: Inheritance (2/3)

(parent) (child)
open ↔ override

- Overriding methods and properties
  - Methods/Properties declared on a superclass that are then redeclared on a derived class must be prefaced with *override* keyword
  - If there is no *open* modifier on a method/property, declaring a method/property with the same signature in a subclass is not allowed

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
    open val count = 2
}

class Circle : Shape() {
    override val count = 0
    override fun draw() { /*...*/ }
//    override fun fill() {/*...*/} (X)

}

class Rectangle : Shape() {
    override val count = 4
}
```

OOP: Inheritance (3/3)

Initialization order

- During the construction of a new instance of a derived class, the base class initialization is done as the first step

```
open class Base(val name: String) {

②    init { Log.d("ITM","Initializing a base class") }

③    open val size: Int =
         name.length.also { Log.d("ITM","Initializing size in the base class: $it") }
}

class Derived(name: String, val lastName: String) :
①  Base(name.replaceFirstChar { it.uppercase() }.also { Log.d("ITM","Argument for the base class: $it") }) {

④    init { Log.d("ITM","Initializing a derived class") }

⑤    override val size: Int =
         (super.size + lastName.length).also { Log.d("ITM","Initializing size in the derived class: $it") }
}
```

Q&A

- Next video
  - Kotlin Basics (Part II & III)