Android App Development with Kotlin

Functional Programming in Kotlin
Data Classes

1

1

Exam I

- Thursday 10/16
- Format
  - Short answer questions
  - Multiple choice
  - True or False
  - Coding
  - 14~15 questions total

2

2

Kinds of Questions to Expect

- Write (simple) Kotlin code
- Explain concepts
- Predict the output of example provided
- Rewrite the example code provided with using specific techniques (e.g., `Elvis` operator, `when`)
- Explain the examples provided
- Distinguish correct vs. incorrect syntax

How to study

- Review examples and code from lecture.

3

3

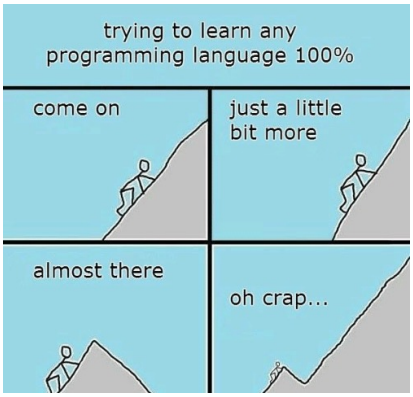
Topics

- Variables/Constants and Types
- Functions, Range/Iteration, When
- Collections, safe-call operator, Elvis operator, Class Any, Type checking, Type Casting
- Class, Inheritance, Abstract class, Interface
- Functional Programming in Kotlin, Data Class

We will move to Android App Development after the exam. Install Android Studio.

4

4



5

5

Functional Programming in Kotlin

6

6

Functional Programming with Lambdas

- Kotlin was born as a **mixed paradigm** language that
  - supports the imperative, OO, and functional paradigms.
  - Kotlin is **multi-paradigm language**
- **Imperative** style is familiar;
  - it is easier to write, due to our familiarity, but is hard to read
  - Imperative style: what + how
- **Functional** style is less familiar;
  - it is harder to write, due to our unfamiliarity, but is easier to read
  - Declarative style: what + **NOT how**

https://kotlinlang.org/docs/lambdas.html

7

7

Q: write **Java (python, JavaScript, or C++)** code to sum the integers 1 through 10.

8

8

Q: write **Java (python, JavaScript, or C++)** code to sum the integers 1 through 10.

```
int total = 0;
for (int i = 1; i <= 10; i++)
   total += i;
System.out.println(total);
```

```
total=0
for i in range(1,11):
   total += i
print(total)
```

```
sum(range(1,11))
```

The computation method is **variable assignment** (emphasizes changes in state)

Summing the integers 1 to 10 in functional Language:

```
sum [1..10]
```

The computation method is **function application**.

9

9

Lambda Expressions

Alonzo Church develops the **lambda calculus**, a simple but powerful theory of functions (1930s)

10

10

Lambda Expressions

- **Turing machines** (by Alan Turing, 1912-1954) are an important model for computation.
- **Lambda (λ) calculus** (by Alonzo Church, 1903 – 1995) has played an important role in the development of the theory of programming languages.
  - **Lambdas are short functions**
  - **Rather than passing data to functions, we can use lambdas to pass a piece of executable code to functions.**

Alonzo Church develops the **lambda calculus**, a simple but powerful theory of functions (1930s)

11

11

- Functions has four parts:

12

12

- Functions has four parts:
  - name, return type, parameters list, and body
- Lambdas carry over only two parts:

λx → x+x

13

13

- Functions has four parts:
  - name, return type, parameters list, and body
- Lambdas carry over only the **parameters list** and **body**

λx → x+x

the **nameless (anonymous) function** that takes a number **x** and returns the result **x+x**.

```
parameter list -> body
```

- The lambda calculus incorporates two simplifications
  - **"anonymous":**
  - **"currying":**

14

14

- Functions has four parts:
  - name, return type, parameters list, and body
- Lambdas carry over only the **parameters list** and **body**

λx → x+x

the **nameless (anonymous) function** that takes a number **x** and returns the result **x+x**.

```
parameter list -> body
```

- The lambda calculus incorporates two simplifications
  - **"anonymous":** without giving them explicit names.
  - **"currying":** takes multiple arguments into a chain of functions each with a single argument (celebrating the work of **Haskell Curry** (1900-1982))

15

15

- A lambda is wrapped within **{}**. The body is separated from the parameter list using a hyphenated arrow **->**

```
{ parameter list -> body }
{ x -> x + 5}
```

Treats functions **"anonymously"**, without giving them explicit names.

```
fun addFive(x: Int) = x + 5
// lambda can have single parameter
{ x: Int -> x + 5}
```

```
fun addInts(x: Int, y: Int) = x + y
// lambda can have multiple parameters
{ x: Int, y: Int -> x + y}
```

```
fun stuff() = "Pow!"
// lambda has no parameter, you can omit ->
{ "Pow!"}
```

16

16

Lambda1.kt

You can assign a lambda to a variable

```
val addFive = {x: Int -> x+5}
// simpler way
println ("Pass 6 to addFive: ${addFive(6)}")
// Execute a lambda's code by invoking it
println ("Pass 6 to addFive: ${addFive.invoke(6)}")

val addInts = {x: Int, y:Int -> x + y}
println ("Pass 6, 7 to addInts: ${addInts(6,7)}")
```

17

17

Lambda expressions have a type

```
// Note that ()'s are required for (Int, Int) -> Int
// It is because of curried function (associate to the right)

val intLambda: (Int, Int) -> Int = {Int: x, Int: y -> x*y}

// The compiler knows that x and y needs to be an Int,
// so we can omit its type

val intLambda: (Int, Int) -> Int = {x, y -> x*y}
println ("Pass 10, 11 to intLambda: ${intLambda(10,11)}")
```

18

18

```
// You can replace a single parameter with "it"
// since compiler can infer its type

val addSeven: (Int) -> Int = {x -> x + 7}

val addSevenSimpler: (Int) -> Int = {it + 7}


// This will NOT work since compiler cannot infer its type

val addSeven = {it + 7}   // No way!

println ("Pass 12 to addSeven: ${addSeven(12)}")
```

19

19

```
// You can replace a single parameter with "it"
// since compiler can infer its type

val addSeven: (Int) -> Int = {x -> x + 7}

val addSevenSimpler: (Int) -> Int = {it + 7}


// This will NOT work since compiler cannot infer its type

val addSeven = {it + 7}   // No way!

println ("Pass 12 to addSeven: ${addSeven(12)}")


// even if the lambda has no parameters,
// its type definition still includes the ()'s

val myLambda: () -> Unit = {println("Hi!")}
myLambda()
```

20

20

- What's the difference?
  - `val x = { "hello" }`
  - `val y = "hello"`

21

21

How to print both "hello" ?

- What's the difference?
  - `val x = { "hello" } //` **Assigns a lambda to x**
  - `val y = "hello"  //` **Assigns a string to x**

22

22

Higher-order function

- A function that use a lambda
  - **as a parameter** or
  - **return value**
  - is known as a higher-order function

23

23

Lambda2.kt

You can pass a lambda to a function

```
fun convert(x: Double, converter: (Double) -> Double): Double {
    val result = converter(x)
    println("$x is converted to $result")
    return result
}
fun main(){
    convert (20.0, {c: Double -> c * 1.8 + 32} )
    convert (20.0, {c -> c * 1.8 + 32} )
    convert (20.0) {c -> c * 1.8 + 32}
    convert (20.0) {it * 1.8 + 32}
}
```

Which one is (or are) correct syntax(es)?

24

24

You can pass a lambda to a function

```
fun convert(x: Double, converter: (Double) -> Double): Double {
    val result = converter(x)
    println("$x is converted to $result")
    return result
}
fun main(){
    convert (20.0, {c: Double -> c * 1.8 + 32} )
    convert (20.0, {c -> c * 1.8 + 32} )
    convert (20.0) {c -> c * 1.8 + 32}
    convert (20.0) {it * 1.8 + 32}
}
```

Which one is (or are) correct syntax(es)?

- We can omit its type if compiler can infer type
- If the final parameter of a function is a lambda, you can move the lambda argument outside the function call's parameters.
- we can use "`it`" if each lambda uses a single parameter whose type the compiler can infer

25

25

```
fun convertFive(converter: (Int) -> Double): Double {
    val result = converter(5)
    println("5 is converted to $result")
    return result
}

fun main(){
    convertFive { it * 1.8 + 32}
}
```

How about this?

26

26

```
fun convertFive(converter: (Int) -> Double): Double {
    val result = converter(5)
    println("5 is converted to $result")
    return result
}

fun main(){
    convertFive { it * 1.8 + 32}
}
```

- You can also remove the ()'s entirely (Note that there are no parenthesis in that function call)
  - This is possible because the function has only one parameter, which is a lambda

Do I really need to know about them all?

27

27

Lambda3.kt

A function can return a lambda

```
fun getConversionLambda(str: String): (Double) -> Double = {
    when(str){
        "CelsiusToFahrenheit" -> it * 1.8 + 32.0
        "KgsToPounds" -> it * 2.204623
        "PoundsToTons" -> it / 2000.0
        else -> it
    }
}
```

```
// You can invoke the lambda returned by a function
getConversionLambda("KgsToPounds")(2.5)

                    OR

// use it as an argument for another function
getConversionLambda("KgsToPounds").invoke(2.5)
```

28

28

A function that received AND returns lambdas

```
typealias DoubleConversion = (Double) -> Double


fun combine(lambda1: DoubleConversion,
            lambda2: DoubleConversion): DoubleConversion{
    return { x:Double -> lambda2( lambda1(x) ) }
}
```

29

29

A function that received AND returns lambdas

```
typealias DoubleConversion = (Double) -> Double


fun combine(lambda1: DoubleConversion,
            lambda2: DoubleConversion): DoubleConversion{
    return { x:Double -> lambda2( lambda1(x) ) }
}

// define two conversion lambda
val kgsToPoundsLambda = getConversionLambda("KgsToPounds")
val poundsToTonsLambda = getConversionLambda("PoundsToTons")

// combine the two lambdas to create a new one
val kgsToTonsLambda =
            combine(kgsToPoundsLambda, poundsToTonsLambda)
```

30

30

31

Built-in higher-order functions

- Kotlin comes with a bunch of built-in higher-order functions.
  - **Too many – refer to documentation**
- Many Kotlin's higher-order functions are designed to **work with collections**.

32

31

32

minBy(), maxBy(), sumOf()

Lambda4.kt

- `min(), max():` works with basic types

```
val ints = listOf(1,2,3,4)

println (ints.maxOrNull() )
```

- `minBy(), maxBy(), sumOf():` higher-order functions

33

```
/*
When two objects have matching property value,
we might want == operator to evaluate to true.
Kotlin developers came up with the concept of a data class
*/
data class Grocery(val name: String, val category: String,
                   val unit: String, val unitPrice: Double,
                   val quantity: Int)


val groceries = listOf (
  Grocery ("Tomatoes", "Vegetable",  "lb",     3.0, 4),
  Grocery ("Mushrooms", "Vegetable", "lb",     4.0, 1),
  Grocery ("Bagels",    "Bakery",    "Pack",   1.5, 2),
  Grocery ("Olive oil", "Pantry",    "Bottle", 6.0, 2),
  Grocery ("Ice Cream", "Frozen",    "Pack",   3.0, 3)
)
```

34

33

34

```
// find the item in groceries with the highest unitPrice
val highestUnitPrice =
      groceries.maxByOrNull { i: Grocery -> i.unitPrice }

// The Lambda has one parameter, so we can just use "it"
val highestUnitPrice =
      groceries.maxByOrNull { it.unitPrice }
println ("highest Unit Price: $highestUnitPrice")

// find the item in groceries with the lowest quantity
val lowestQuantity = groceries.minByOrNull { it.quantity }
println ("lowest Quantity: $lowestQuantity")
```

```
// sumOf returns a sum of the items in a collection
val sumQuantity = groceries.sumOf{ it.quantity }

// return type of sumOf is Double
val totalPrice=groceries.sumOf{ it.quantity*it.unitPrice }
```

35

filter(), map(), forEach()

Lambda5.kt

- `filter():` search or filter a collection and returns a new list
- `map():` returns a List
- `forEach():` is a function, so you can use it in function call chains

```
// returns a List containing those items
// from groceries whose category value is "Vegetable"
// returning entire row

val vegetables = groceries.filter {it.category == "Vegetable"}


// There's whole FAMILY of filter functions
// filterNot returns those items
// where the lambda body evaluates to false

val notFrozen = groceries.filterNot {it.category == "Frozen"}
```

36

35

36

```
// The map function returns a List
// There are many variations of the map functions

// this example creates a new List,
// with the name of each Grocery item in groceries
// returns the specific column only, not entire row

val groceryNames = groceries.map {it.name}

// this returns a List containing each unitPrice * 0.5

val halfUnitPrice = groceries.map {it.unitPrice * 0.5}

// Calls filter function first, then
// calls map on the returning list

val newPrices =
      groceries.filter {it.unitPrice > 3.0}.map { it.unitPrice * 2 }
```

37

37

```
println("Grocery names:")
groceries.forEach { println(it.name) }
```

```
// equivalent
for (item in groceries) println(item.name)

// equivalent
groceries.map {it.name}
```

```
println("\nGroceries with unitPrice > 3.0: ")
groceries.filter { it.unitPrice > 3.0 }.forEach { println (it.name) }
```

```
// equivalent
for (item in groceries){
    if (item.unitPrice > 3.0)
        println(item.name)
}
```

38

38

```
// local variable that the lambda can access

var itemNames = ""
groceries.forEach { itemNames += "${it.name},  " }
```

```
// equivalent
for (item in groceries)
     itemNames += "${item.name}"
```

```
Lambda's closure:
The local variables defined outside that the lambda can access
In this example: itemNames
```

39

39

groupBy()

Lambda6.kt

- `groupBy:` group the items in your collection - same as SQL group by
- `fold:` specify initial value and perform some operation on it for each it...

```
// groceries.groupBy {it.category } means
// "group each item in groceries by its category value"

// groupBy returns a Map
// key: each criterion, value: List of items

groceries.groupBy {it.category }.forEach {
    //prints the Map key (Grocery category)
    println(it.key)

    //gets the corresponding value: List<Groceries>,
    //for the Map's key
    it.value.forEach { println ("  ${it.name}") }
}
```

```
Vegetable
   Tomatoes
   Mushrooms
Bakery
   Bagels
Pantry
   Olive oil
Frozen
   Ice Cream
```

40

40

fold()

- `fold` function - **one of the flexible higher-order functions**
- take two parameters: **initial value** and **operation**
- specify **initial value** and perform some **operation** on it for each item

```
val ints = listOf(1, 2, 3, 4)
val sumOfInts =
         ints.fold(0) { runningSum, item -> runningSum + item}

println ("sumOfInts: $sumOfInts") // 1+2+3+4
```

1. creates a variable named `runningSum` which is initialized with 0.
2. The function takes the value of the first item in the collection (1 in this example) and adds it to `runningSum`
3. The function add 2 to `runningSum` (now `runningSum` is 3)
4. The function add 3 to `runningSum` (now `runningSum` is 6)
5. The function add 4 to `runningSum` (now `runningSum` is 10)
6. The function returns the final value of `runningSum` to `sumOfInts`

```
sum Of Ints: 10
```

41

41

```
val names =
 groceries.fold ("") {str, item -> str + "  ${item.name}"}

println ("names: $names")
```

```
names:   Tomatoes  Mushrooms  Bagels  Olive oil  Ice Cream
```

```
val changeFrom50 = groceries.fold (50.0)
 {change,item -> change - item.unitPrice * item.quantity}

println ("changeFrom50: $changeFrom50")
```

```
changeFrom50: 10.0
```

42

42

In-class exercise (1/2)

1. Convert the following Kotlin code using `filter` and `map` functions

```
fun main() {
    var doubleOfEven = mutableListOf<Int>()
    for (i in 1..100)
        if (i % 2 == 0)
            doubleOfEven.add(i * 2)

    println(doubleOfEven)
}
```

43

43

In-class exercise (2/2)

2. Convert the following Kotlin code **using higher-order (lambda) function none**

```
fun isPrime(n:Int): Boolean{
    if (n < 2) return false
    for (i in 2 until n){
        if (n % i == 0)
            return false
    }
    return true
}

println(isPrime(1)) //false
println(isPrime(2)) //true
println(isPrime(3)) //true
println(isPrime(4)) //false
```

```
// none returns true if no elements match the given predicate.
// none returns false if lambda (predicate) returns true
```

https://kotlinlang.org/api/core/kotlin-stdlib/kotlin.collections/none.html

44

44