ATLAS 4320: Advanced Mobile Application Development

Into to Kotlin

Kotlin Background

Kotlin is a language that was created by JetBrains (the same company behind intelliJ) in 2010. Many people think of Kotlin as a better, or more streamlined, version of Java. While this is somewhat true, and Kotlin is 1**00% interoperable with Java** (code base could be a mix of Java and Kotlin, much like Swift and ObjC), Kotlin is a language in its own write that has exponentially grown in popularity after Google declared it as the preferred language for Android development in 2019.

Some of the advantages of using Kotlin over Java include the following:

- Inferred Types
    - This cuts back on the amount of code needed and Kotlin does a pretty good job of guess what you want based on the context
- Nil Safety
    - The lack of this feature in Java is one of its major flaws and the source of many runtime errors
    - Works a lot like optionals in swift in that you have to add a question mark to the end of the type when you declare it if you want it to be able to hold a null value
- Semicolons are optional
    - Hell yeah! Only need semicolons for consecutive statements on a single line
- Better handling of async/sync operations using **coroutines**
    - Works similar to async/await in JavaScript
- Functional Programming
    - Functions do not need to be class members as they do in Java
    - Also get higher-order functions (take functions as parameters or return functions), anonymous functions, lambdas, closures, etc.
- And more!

Demo

I'll be using IntelliJ IDEA (Community Edition) to run through my Kotlin intro, but you can use any IDE that supports Kotlin such as Atom or Sublime. Also, you can use the online playground on the Kotlin website to write and run Kotlin code: https://play.kotlinlang.org/. This is the only time we'll be using an IDE besides Android Studio, so if you don't want to mess with setting up an IDE then I recommend just using the playground.

Setup Project in IntelliJ

Select File -> New
Select *Kotlin* for the sidebar on the left of the dialog and then *JVM* | *Idea* from the main section
Choose a name and location for your project and **select** Finish

In the project navigator pane, **ctrl-click** on the src folder and select *New -> Kotlin File/Class*
Give it a name and **press** enter

We need to setup a `main()` function to give our program an entry point

```
fun main() {
}
```

First things first, let's take a look at variables, constants, and basic types

```
//region: constants, variables, and types

//region: var and val
//variables (implicit type of Int)
var myVar = 2 //implicit type (int)
println(myVar)
myVar += 2
println(myVar)

//constants use val keyword
val myConst = 2 //implicit type (int)
println(myConst)
//can't change a constant! next line won't compile
//myConst += 2

//endregion


//region: basic types (explicitly typed)

//Integers
val myInt: Int = 4
//for larger numbers you can use underscores to make them more readable
val oneBillion = 1_000_000_000
println(oneBillion)
//double (default inferred for floating-point numbers
val myDouble: Double = 4.50
//float (if you specifically need a Float you must explicitly ask for it by
appending "f" to the number like so)
val myFloat: Float = 4.5f
```

```
//Booleans
var myBool = true
myBool = false
println(myBool)

//Chars - single quotes
var myChar = 'I'
//use backslash to escape special characters (standard practice)
var myNewLine = '\n'
println("$myNewLine $myChar $myChar $myNewLine $myChar")

//Arrays - immutable size, mutable data, can be mixed type
//array of ints
val myIntArray = arrayOf(1,2,3,5)
//accessing and modifying using getters and setters
println(myIntArray.get(1))
myIntArray.set(1, 10)
//access using index
println(myIntArray[1])

//array of mixed type
var myCharArray = arrayOf('I', 'S', 'A', 'A', 'C', "Sheets", 22)

//Strings
var myName = "Isaac Sheets"
myName = "Isaac C. Sheets"
println(myName)

//string literal
//raw string (preserve white space and new lines)
val message = """
Hi there!
    This is a message
        From:    Isaac
"""//.trimIndent()
println(message)

//string template (variables or expressions in strings)
val template = "My name is $myName and it is ${myName.length} characters long"
println(template)

//endregion

//region: nullable types and checking
//must have explicit type with question mark to hold a null value
var score: Int? = null
```

```
//null check (fails)
if (score != null) {
    println("Score: ${score.minus(10)}")
} else {
    println("score is null")
}

//give score a value
score = 80

//null check (succeeds)
if (score != null) {
    //compiler remembers that we checked null so we can access methods/properties
of score inside this block
    println("Score: ${score.minus(10)}")
}

score = null

//safe call (returns result or null)
println("Safe Call")
println(score?.minus(10)) //can chain as many safe calls together as necessary

//elvis operator -- or default value
println("Elvis Operator:")
println(score?.minus(10) ?: 10)

//!! operator -- force not null assertion -- BAD
//println("Score: ${score!!.toString()}") //causes Runtime Error - Null Pointer
Exception :(
//endregion

//endregion
```

Type Checks and Casting

```
//region: TYPE CHECKS AND CASTING

//is keyword -- checks type and automatically casts in most cases
var mixedArray = arrayOf("Test string", 'C', 1_000, 1.00)

fun typeChecker(x: Any) {
    if(x is String) {
        println("String has length ${x.length}") //x is automatically cast to
```

```
string
    }
    else {
        println("$x is not a string")
    }
}

//check types of each element in array
mixedArray.map{typeChecker(it)}

//as (unsafe) and as? (safe) keyword to cast to specific type
for(item in mixedArray) {
    var cast = item as? String //change to as and see what happens
    if(cast != null) {
        println("successfully cast $cast as String")
    }
}

//endregion
```

Loops

```
//region: LOOPS and EXPRESSIONS

//RANGES
//Kotlin uses ".." syntax and
for(i in 1..5) println(i)
println()
for(i in 1 until 5) println(i)
println()
for(i in 1..5 step 2) println(i)
println()
for(i in 5 downTo 1) println(i)
println()


//WHEN EXPRESSIONS
val myScore = 80
val highScore = 120

//replaces a C style switch statement
when(myScore) {
    in 0 until highScore -> println("current score less than high score")
    highScore -> println("new high score!")
    else -> println("above high score")
}
```

```
println()

//FOR LOOPS
val fruits = arrayOf("Apples", "Bananas", "Strawberries", "Cherries",
"Raspberries")

//basic syntax
for(fruit in fruits) {
    println("Current fruit: $fruit")
}
println()

//get the index of the iterator
for((i, fruit) in fruits.withIndex()) {
    println("Fruit #$i: $fruit")
}

println()

//use indices
for(i in fruits.indices) {
    println("Fruit #$i: ${fruits[i]}")
}
println()

//WHILE LOOPS
//nothing new here
var loop = 5
while(loop > 0) {
    println(loop)
    loop--
}

//endregion
```

Collections

Kotlin has a standard concept of Collections which are just groups of items. It has the three basic type present in most programming languages

- List - stored in order
- Set - contains only unique numbers
- Map (or Dictionary) - key/value pairs

There are two types of collections: mutable and immutable. As the names suggest, you can change mutable collections but immutable collections are read only after they are created.

```
//region: COLLECTIONS
```

```
//LIST
//immutable (can only perform read operations)
val myList = listOf(1,2,3,4,5,6,5,4,3,2,1)
//myList[0] = 10 //won't compile since this list is read-only
println(myList[0])

//mutable list (can perform read AND write operations)
val myMutableList = mutableListOf(1,2,3,4,5,6)
myMutableList[0] = 10
println(myMutableList[0])

//SETS - not necessarily ordered
//immutable
val mySet = setOf(1,2,3,4,5,6,6,6,6) // create mutable set using mutableSetOf()
mySet.map{println(it)}

//list to set
val uniqueList = myList.toSet()
myList.map{print(it)}
println()
uniqueList.map{print(it)}
println('\n')

//MAPS
var myMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
myMap["four"] = 4

for(pair in myMap.entries) {
    println("${pair.key} to ${pair.value}")
}

//endregion
```

Functions and Lambdas

Functions in Kotlin are first-class which means they can be assigned to a variable or other data structure and get passed around to higher-order functions. If you're not familiar, a higher order function is just a function that either takes another function as a parameter or returns a function itself (or both). This is a necessary feature of a functional programming language like Kotlin.

Kotlin also has lambda and anonymous functions as options for function literals — these are just functions that are not declared and instead are passed immediately as an expression.

```
//region: FUNCTIONS and LAMBDAS
```

```
//NOTE: these are all local scope functions meaning they are functions within a
function (inside of the main() function)
//syntax for member functions (inside classes) and top level functions (not inside
anything) is the same

//basic syntax = "fun functionName(parameter1: Type, parameter2: Type): ReturnType
{}
fun add(x: Int, y: Int): Int {
    println("Adding $x + $y")
    return x+y
}

//single expression function with implicit return type
fun quickAdd(x: Int, y: Int) = x + y

println(add(2,2))
println(quickAdd(3,3))

//default, named arguments
fun printGreeting(name: String, senderName: String = "A friend", message: String =
"" ) {
    println("Hello $name! $message From: $senderName")
}

printGreeting("Bob")
printGreeting(name = "Bill", message = "Hope you're well.")
printGreeting(name = "mary", senderName = "Isaac")

println()

//LAMBDAS
//functions that are not declared but passed immediately as expressions
val nums = setOf(1, 6, 3, 22, 10_000, 0, -300, 100, 355, 87)
//filter the list to only have items greater than 100
var highNums = nums.filter {item -> item >= 100 }


//printing using the map() function is actually another example of a lambda!
nums.map { print("$it ,") }
println()
highNums.map {print("$it ,")}

//alternate syntax for lambdas
//named lambda

//named lambda (longest syntax, same result)
val filterLam: (Int) -> Boolean = { item: Int -> item >= 100 }
```

```
highNums = nums.filter(filterLam)

//using implicit parameter name (works when it only takes a single parameter)
//shortest syntax, same result
highNums = nums.filter { it >= 100 }

//endregion
```

Classes

Not much is new here for regular classes. Data classes are a Kotlin specific concept that aim to make it easier to create classes that only represent data (no real functionality). Tells Kotlin to treat the object as a Data Transfer Object. We get a better string representation, better equals functionality, easier copies, and easy destructuring built in.

```
//plain class, no constructor
class Instrument {

}
//empty instance
var myInstrument = Instrument()

//class with primary constructor
open class Food constructor (var name: String, var calories: Int) {

    //private property
    var rating: Double? = null

    //public method
    fun eat() {
        println("eating $name, yummy yum")
    }

    //init
    init {
        println("init your $name!")
    }



    //secondary constructor to add a rating
    constructor(name: String, calories: Int, rating: Double) : this(name, calories)
{
        this.rating = rating
    }

    //override the toString()
```

```
    override fun toString(): String {
        return "$name, $calories, ${rating ?: "no rating"}"
    }
}
//oatmeal instance
var oatmeal = Food("oatmeal", 500)
//hummus instance
var carrot = Food("carrot", 50, 8.0)

println(oatmeal)
println(carrot)

carrot.eat()

//inheritance (add open keyword to parent class)
class Snack(name: String, calories: Int, var category: String = "Snack") :
Food(name, calories) {

    //secondary constructor for rating functionality
    constructor(name: String, calories: Int, category: String = "Snack", rating:
Double): this(name,calories,category) {
        this.rating = rating
    }
    init {
        println("making a fun snack: $name")
    }

}

var hummus = Snack("Hummus", 200, rating = 10.0)

hummus.eat()
println(hummus)

println()

//DATA CLASSES

data class User(val id: Int, val name: String, val email: String)

var user1 = User(1,"John", "password123")
var copyUser1 = User(1, "John", "password123")

println(user1)
println(user1 == copyUser1)

//destructure
```

```
val (id, name, email) = user1

//endregion
```

Coroutines

Supported via an external library (supported by JetBrains), Kotlin Coroutines are essentially light-weight threads. For now, we'll just take a high level look at them so you can familiarize yourself with them. We'll dive deeper once we start working on async tasks.

```
//region: COROUTINES
//basic example
println("before launching coroutine")

GlobalScope.launch { // new coroutine in background
    println("in coroutine")
    delay(1000) //non-blocking suspension of coroutine execution
    println("coroutine done")
}

println("Coroutine launched, waiting for result")
Thread.sleep(2000L) //sleep the execution of the application so it keeps everything
alive until coroutine finishes

println()
//better example
runBlocking{ //start a new blocking coroutine on the current thread (main in our
case)
    var asyncJob = GlobalScope.launch { // new coroutine in background
        println("starting coroutine code")
        delay(1000) //non-blocking suspension of coroutine execution
        println("coroutine done")
    }

    println("launched coroutine")
    asyncJob.join()
    println("continuing execution")
}

//endregion
```

Exceptions

Exceptions in Kotlin work exactly the same way as they do in Java. There are a variety of built in JVM exceptions that you can use, or you can create your own custom exception with a custom name and message. We'll look at both.

```kotlin
//region: EXCEPTIONS

//custom type
class MathError(message: String): Exception(message)

fun divide(dividend: Double, divisor: Double): Double {

    if (divisor == 0.0) {
        throw MathError("Can't divide by 0!")
    }
    else {
        return dividend/divisor
    }

}

try {
    val result = divide(50.0, 0.0)
    println("Successful division: $result")
} catch (e: Exception){
    println("Bad division! ${e.toString()}")
} finally {
    println("I'm done trying")
}
//endregion
```