Kotlin For Android

Workbook

Basic Syntax (presentation, video)

Basic Types

Kotlin has no primitive types. All java primitives have corresponding immutable types. In Kotlin everything is an instance of some type.

Kotlin	Java
`Boolean`	`boolean, Boolean`
`Int`	`int, Integer`
Same for `char, byte, short, long, float` and `double`	
`Any`	`Object`
`Unit`	`void`
`Nothing`	`void`
`Array<Int>`	`Integer[]`
`IntArray`	`int[]`
Same for `BooleanArray, CharArray, ByteArray, ...`	

Comments are the same as java - //, /* */, /** */

Defining Packages

Package specification should be at the top of the source file followed by import statements (same as Java):

```
package my.demo

import android.view.View
import android.view.View.VISIBLE
import my.demo.Contract.View as CView
```

There are no static imports. Just import the class member without the `static` keyword and it will work.

As shown above it is possible to alias conflicting class names to avoid using fully-qualified class names in your code.

Variables

There is no `final` keyword in Kotlin. Instead variables are declared as `val` or `var`. `val` is an immutable reference and cannot be reassigned, `var` can be reassigned. Compiler always knows the type the right hand side of an assignment operation is and so it can always infer the correct type to use on the left hand side. Use this type inference carefully and only for very obvious types.

```
// Java
final int a = 1;
int b = 2;

// Kotlin
val a = 1
var b = 2
```

Constants in Kotlin are denoted with the extra keyword `const`, which is stricter than Java's `static` and is limited to scalar values (numbers), strings and arrays.

```
const val ONE = 1
const val FOO = "FOO"
const val FOO1 = "FOO" + ONE
```

Strings

All strings literals are templates and will evaluate expressions.

```
"Hello World!"
"Hello $name!"
"Hello ${user.name}!"

println("$1 \$bills")          // "$1 $bills"
println("""$1 ${'$'}bills""") // "$1 $bills"
```

Variable expansion is more performant than `String.format()`.

Defining Functions

What we call methods in Java are called functions in Kotlin. Functions are first-class types in Kotlin and can be defined anywhere in the code base, also outside of classes and interfaces or inside other functions, same as any other type

We can also pass functions as parameters to other functions. Functions that receive or return functions are usually referred to as higher-order functions.

Because of this a keyword is required to denote the beginning of the declaration of a new function and that is `fun`. This is not a pun.

```
fun world() = "World"
fun main(args: Array<String>) {
    fun hello(who: () -> String) = println("Hello ${who()}!")
    hello(::world)
}
```

The definition of function parameters is reversed from Java. The parameter name comes first, followed by a colon and the type of the parameter.

There are 2 types of function bodies

- a normal body which contains 0 to any number of expressions and is denoted the same as java - between `{}`
- an expression body which starts with an equals sign "`=`" followed by a single expression. As the name suggest it contains one and only one expression.

Defining Interfaces

Interface definition is very similar to java, except that the `extends` keyword is replaced by a single colon "`:`".

```
interface Runnable {
    fun run()
}

interface RepeatingRunnable : Runnable {
    fun run(times: Int) {
        (1..times).forEach { run() }
    }
}
```

Defining Classes

A class declaration is similar to interfaces just with the keyword `class`. If a class extends from another then the parent class is the first type on the right-hand side of the "`:`". There is no `implements` keyword and interfaces are comma separated after the parent class.

```
open class Greeting : Runnable {
    override fun run() = println("Hi!")
}

class RepeatingGreeting : Greeting(), RepeatingRunnable
```

If the class has no implementation you can skip the braces.

Coding Tasks

Clone the github repo and build it on the command line (3-4 min) to make sure you have all the dependencies and the right version of gradle.

```
git clone git@git.yelpcorp.com:kotlin-demo
cd kotlin-demo
export ANDROID_HOME=$HOME/Library/Android/sdk
echo "sdk.dir=$HOME/Library/Android/sdk" > local.properties
./gradlew assemble
```

In Android Studio use “Import Project (Gradle….”) to add the repo to Android Studio
In the main activity add a single `TextView` showing “Hello World!”
Write an espresso test to make sure the text is visible.

Bonus Task

Get the `String` that the activity shows via an object.
Mock the string for your tests.

Constructors and Control Flow (presentation, video)

Constructors

A class in Kotlin can have a **primary constructor** and one or more **secondary constructors**. The primary constructor is part of the class header: it goes after the class name (and optional generics).

```
class Person @Inject private constructor(firstName: String)
```

If the constructor has no annotations or visibility modifiers then the keyword `constructor` can be dropped.

```
class Person(firstName: String)
```

The primary constructor has no body. Any initialization code must be places inside `init` blocks. `init` blocks and property initialization expressions are executed in the order in which they are defined. Here’s a little example - try to guess the order in which the `println` statements will be executed. There is an answer to compare against in the git repo.

```
open class Parent(arg: Unit = println("Parent primary default argument")) {
    private val a = println("Parent.a")

    init {
        println("Parent.init")
    }

    constructor(
            arg: Int,
            arg2: Unit = println("Parent secondary default argument")
    ) : this() {
        println("Parent secondary constructor")
    }

    private val b = println("Parent.b")
}

class Child(arg: Unit = println("Child primary default argument")) : Parent(2) {
    val a = println("Child.a")

    init {
        println("Child.init 1")
    }

    val b = println("Child.b")

    constructor(
            arg: Int,
            arg2: Unit = println("Child secondary default argument")
    ) : this() {
       println("Child secondary constructor")
    }

    init {
        println("Child.init 2")
    }
}

Child(2)
```

Defining Objects

An object is a Singleton class. All of its members are accessed via its name.

```
object StringUtils {
   fun isEmpty(string: String?) = string.orEmpty().length == 0
}

StringUtils.isEmpty("")
```

Additionally every normal class has a special `object` called the `companion object`. The companion object is also accessed via the enclosing class' name. This provides a way to add static members to a class, via the `@JvmStatic` and `@JvmField` annotations.

```
class Foo {
   companion object {
       const val BAR = "BAR"
       @JvmStatic fun newFoo(): Foo = Foo()
   }
}
```

```
// kotlin
val bar = Foo.BAR
val foo = Foo.newFoo()

// java
String bar = Foo.Companion.BAR;
Foo foo = Foo.Companion.newFoo();

// java after annotations
String bar = Foo.BAR;
Foo foo = Foo.newFoo();
```

Control Flow

There is no `goto` in Kotlin and the only 3 expressions that can alter the flow of executions are:

- `return`
- `break`
- `continue`

You can mark the entry of a loop or a function with a label and later on use of the above expressions with the given label to jump a few levels in the execution hierarchy

```
loop@ for (i in 1..100) {
   for (j in 1..100) {
       if (i==50 && j == 70) break@loop
   }
}
```

`while` - works the same as Java
`for` - can be used only with iterables.
`if` - same as java, but can also be used as an expression.

```
val x = if (Math.random().toInt() % 2 == 0) "even" else "odd"
```

`when` - replaces Java's `switch`

- Can also be used as an expression
- Branches are exclusive and cannot fall-through to the next branch
- `else` instead of `default` case
- MUST be exhaustive if used as an expression and will fail to compile otherwise

```
val z = when {
   x.isOdd() -> "x is odd"
   x.isEven() -> "x is even"
   else -> "x is funny"
}
```

Modality and Visibility

The default modality of classes, properties and functions is `final` and the `final` keyword does not exist in Kotlin. Instead Kotlin defines the opposite keyword `open` which negates the final modality.

The default visibility in Kotlin is `public`. `public`, `protected` and `private` are the same as in Java. The default Java visibility does not exist in Kotlin and is replaced with the new `internal` visibility. Internally visible members can be accessed only from other members in the same unit of compilation (in gradle that's a module).

Coding Tasks

Split your layout into a fragment.
Create the Fragment via a `newInstance()` companion object function.
Have the Hello World text view be part of the Fragment.
Make sure your espresso test is still passing.

Bonus Task

Play around with fancy animations and transitions.
Explore the android core-ktx extensions for easier work with animations.

Functions (presentation, video)

Operators

Almost every operator and some soft keywords in Kotlin translate into functions marked with the `operator` keyword. Here's a list of all operator functions.

Expression	Translated to
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`
`a++ (++a)`	`a.inc()`
`a-- (--a)`	`a.dec()`
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.times(b)`
`a / b`	`a.div(b)`
`a..b`	`a.rangeTo(b)`
`a in b`	`b.contains(a)`
`a !in b`	`!b.contains(a)`
`a[i]`	`a.get(i)`
`a[i, j]`	`a.get(i, j)`
`a[i_1, ..., i_n]`	`a.get(i_1, ..., i_n)`
`a[i] = b`	`a.set(i, b)`
`a[i, j] = b`	`a.set(i, j, b)`
`a[i_1, ..., i_n] = b`	`a.set(i_1, ..., i_n, b)`
`a()`	`a.invoke()`

`a(i)`	`a.invoke(i)`
`a(i, j)`	`a.invoke(i, j)`
`a(i_1, ..., i_n)`	`a.invoke(i_1, ..., i_n)`
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a *= b`	`a.timesAssign(b)`
`a /= b`	`a.divAssign(b)`
`a == b`	`a?.equals(b) ?: (b === null)`
`a != b`	`!(a?.equals(b) ?: (b === null))`
`a > b`	`a.compareTo(b) > 0`
`a < b`	`a.compareTo(b) < 0`
`a >= b`	`a.compareTo(b) >= 0`
`a <= b`	`a.compareTo(b) <= 0`

As Types

Functions are real types in Kotlin and a class may extend from a function. An example can be seen in the code below.

```
class World : () -> String {
    override fun invoke() = "World"
}

fun hello(who: () -> String) = println("Hello ${who()}!")
hello(World())    // prints "Hello World!"
```

This is very awkward to read and understand and can be improved a little by use of a `typealias`. Type aliases exist only at compile-time and are used mostly for readability when a type is better described in the current context under a different name similar to the `as` keyword in the import statements.

```
fun device() = "Android"

typealias NameProvider = () -> String
typealias DeviceName = () -> String

class World : NameProvider {
   override fun invoke() = "World"
}

fun hello(who: NameProvider, what: DeviceName) {
   println("Hello ${who()} on ${what()}!")
}

hello(World(), ::device) // prints "Hello World on Android!"
hello(::device, World()) // prints "Hello Android on World!"
hello(what = ::device, who = World()) // prints "Hello World on Android!"
```

However be very careful as aliases are just syntactic sugar and anything that satisfies the static typing of the language will still compile. Notice for example the second last line in the above example.

Default and Named Parameters

The parameter names in Kotlin are very important, because you can use them when you call the function and use them to reorder the arguments. This is particularly useful if you have a few default parameters.

```
import java.util.Date

fun hello(
       name: String = "World",
       device: String = "Android",
       time: String = Date().toString()
) = println("Hello $name on $device at $time!")

hello()
hello(device = "Watch")
hello(time = "beer o'clock")
```

Functions With Receivers

When declaring a function you can optionally declare a type on the left-hand side of the function name and separate it by a dot. You can declare such functions as parameters or return types as well as after the keywork `fun`.

A function with receiver declared with the keyword `fun` is more commonly known as an extension function because it's use looks like one is extending the base functionality of the receiver type.

Using the `this` keyword inside the body of a function with receiver gives you access to the instance of the receiver type.

Here is an example:

```
operator fun ViewGroup.plus(@LayoutRes layout: Int): ViewGroup {
   return LayoutInflater.from(context).inflate(layout, this, false)
}

val scrollView = container + R.layout.scroll_view
```

We can also declare generics on the receiver type which will further narrow down the scope in which we can use the function.

```
fun Iterable<Int>.sum() = this.sumBy { it }
println((1..10).sum())                 // prints 55
println(listOf("a", "b", "c").sum()) // does not compile
```

Higher order functions

Kotlin provides improved call-site syntax for functions which accept other functions as their last argument. Because of this it is also rare that functions accept more than one function argument. Here’s the example we already looked at for the syntax when calling a function that accepts another function.

```
fun hello(who: () -> String) = println("Hello ${who()}!")
hello { "World" }
```

Vararg functions

One of the function parameters can be use variable number of arguments (`vararg`). This parameter is denoted with the keyword `vararg`. The position of the `vararg` is not enforced by the compiler, but typically it is the last argument. The only exception being higher-order functions which accept another function as argument. In that case the argument function is last for better call-site syntax.

Inside the function the `vararg` is accessible as an array. If you want to pass an existing array into a `vararg` (or part of a `vararg`) you have to use the spread operator ‘`*`’.

```
val array = arrayOf(1, 2, 3)
val list = listOf(0, *array, 4)
```

Infix functions

If you use the `infix` keyword in a function declaration the function can be used as an operator. The most commonly-used and widely known infix function is the function `to`.

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

They are not necessarily extension functions. Member functions can also be marked as `infix`.

Inline Functions

Inline function come in handy when we want to reuse some piece of code, but don't necessarily want to incur the overhead of calling a method or creating lambdas. The compiler will remove functions marked with the keyword `inline` and instead replace their call-site with the body of the function statically resolving any generics.

Because generics are being resolved if you use an invariant type as generic you can also use an additional keyword `reified` in your generics definition block. This allows you to use type inference actions with the generic - for example casting, or getting its `::class`, which are not possible with normal java generics because of type erasure - that is at run-time the type information of the generics is lost.

Runtime type erasure is the reason why some Java method require us to pass in the instance of the class as an argument in order to use them. Mockito is probably a very common example.

```
// Java
ViewModel model = mock(ViewModel.class)

// Kotlin Before
val model: ViewModel = mock(ViewModel::class.java)
inline fun <reified T> mock() = Mockito.mock(T::class.java) as T
// After
val model: ViewModel = mock()
```

Unfortunately `inline` function are not visible from Java classes, because the Kotlin compiler doesn't compile our Java classes and can't inline bytecode into them.

Java Interoperability

`@JvmName` and `@file:JvmName` can be used to change the name of functions or the synthesized class names for top level functions to provide better Java interop.
`@JvmStatic` can be used to make an object function declaration `static` for Java.

Coding Tasks

Add a SearchView widget.
Update the text of the TextView with the query from the SearchView.
Write an espresso test to make sure the text you input is visible in the TextView.

Bonus Task

Try to implement a subclass of TextView with a `var text` which behaves like the synthetic one from the original Java class.

Properties, Nullability and Annotations (presentation, video)

Java Synthetic Properties

Non-void getter methods from Java classes can be accessed as `val` in Kotlin. Additionally if the class has a matching named setter with a single argument matching the return type of the getter than you access both methods as a single `var` in Kotlin.

```
String getString()   // Java getter
val string: String   // Kotlin val

CharSequence getText()          // Java getter
void setText(CharSequence sss)  // Java setter
var text: CharSequence          // Kotlin var
```

Kotlin Properties

No simple field declarations in Kotlin, instead everything is a property. Declaration is identical to local variables. Interfaces can also declare properties. `var` replaces regular fields, `val` - final fields.

A single property when compiled to java is translated into a private field with a getter that matches the visibility of the property and a setter (for `var` only) that defaults to the same visibility, but can be constraint to a tighter visibility.

```
interface Factory<T: Any> {
    var mock: T?
}
class Provider<T: Any>(init: () -> T): Factory<T> {
    override var mock: T? = null

    @get:JvmName("it")
    val instance: T by lazy(init)
        private set
}
```

Because the only way to access a property is via its getter and setter methods and you can override methods in Java you can also declare properties in interfaces, which simply means you have to create the appropriate getters and setters.

Annotations and Qualifiers

Annotations work same as Java. One addition is that you can have a qualifier for the annotation, because some Kotlin constructs translate into multiple annotation targets in Java (like the properties, which can translate into up to 6 annotation targets). Here's all the qualifiers and which target will be used if you use the qualifier.

Qualifier	`@Foo` **applies to**
`@file:Foo`	Entire .kt file
`@param:Foo`	Parameter of a single-param `fun`
`@receiver:Foo`	Receiver of a `fun` with receiver
Below are exclusive or most relevant to properties	
`@property:Foo`	The property ¯_(ツ)_/¯
`@field:Foo`	Field of a property
`@get:Foo`	Getter of a property
`@set:Foo`	Setter of a property
`@setparam:Foo`	The parameter of the setter of a property
`@delegate:Foo`	The delegate of a property

Property Delegates

You can also delegate the implementation of a property to another instance. The target instance has to implement 2 operation functions for var and only the getter for val.

```
interface Delegate<T> {
    operator fun getValue(
            thisRef: Any?,
            property: KProperty<T>): T

    operator fun setValue(
            thisRef: Any?,
            property: KProperty<T>,
            value: T)
}
```

Below is the syntax for declaring a delegate and the code that the compiler will generate for you if you do.

```
class C {
    var p: T by MyDelegate()
}

class C {
    private val p$delegate = MyDelegate()
    var p: T
```

```
        get() = p$delegate.getValue(this, this::p)
        set(v: T) = p$delegate.setValue(this, this::p, v)
}
```

`lateinit`

A keyword reserved for `var`. It tells the compiler to stop complaining about uninitialized vars during the construction of a class. The compiler will not generate any guards around the property, so if it is a non-null var and you try to access it before your code initialized it you will get an NPE - same as Java.

It is useful in places where you know for sure you will create the instances before you use them, but it is impossible to create them together with the class itself (like Activities, Fragments, etc in Android). You typically create things in the `onCreate()` method instead of inside constructors.

Nullability

Every type has a nullable counterpart. `Any` vs `Any?`. In terms of type hierarchy the 2 types are completely independent and do not inherit for each other. The nullable variants exist only during compile time and are replaced with runtime checks preceding their usage. So at runtime only the non-null versions exist. That's why it is impossible to use reflection on the nullable types.

However it is possible and indeed used throughout the Kotlin standard lib to create an extension function on a nullable type. This helps greatly when dealing with nullablility and can reduce long safe invocation call chains to a much shorter ones.

```
val any: Any = Any()
val maybe: Any? = null

fun show(maybe: Any?) = print(maybe)
fun reallyShow(any: Any) = print(any)

show(maybe)
show(any)

reallyShow(any)
reallyShow(maybe) // compiler error
```

Coding Tasks

Create a BusinessRepo class and a Business data class.
Create a search(query: String) function in the repo that will return an RxJava Single with a list of Business objects.
Adapt your test to still pass.

Bonus Task

Extend the Business data class to cover the full Yelp Public API.

Kotlin Std-Lib Extension Functions (presentation, video)

https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/index.html#functions

Coding Tasks

Create a BusinessSearchResponse data class according to Yelp's Public API.
Create a Retrofit API Interface.
Replace the dummy implementation of BusinessRepo.search(query: String) with a call to Retrofit.
Adapt your tests to still pass.

Bonus Task

Create more data classes from the API and go deeper - maybe a business details page?

Kotlin Std-Lib Collections Extension Functions (presentation, video)

https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html

Coding Tasks

Create a RecylerView and an adapter and display the results from the API into a list.
Adapt your tests to still pass.

Bonus Task

React to clicks and go to a business details page!

Kotlin DSL and Closing Q&A (presentation, video)