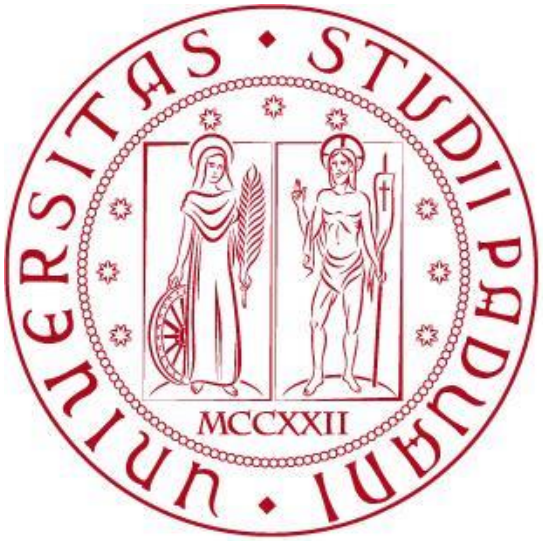

UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria
dell'Informazione

Corso di Laurea in
Ingegneria Informatica

Programmazione di Sistemi Embedded A.A. 2017/2018

KOTLIN: IL FUTURO DELLA PROGRAMMAZIONE ANDROID

A cura di:
Stefano Ivancich
Cristian Lazzarin
Federico Mantovan

Docente:
Carlo Fantozzi

INDICE

Sommario

INTRODUZIONE

Kotlin è un linguaggio di programmazione ad oggetti open-source sviluppato dall'azienda Jet-Brains, la stessa che ha prodotto IDE rinomati come Android Studio. Questo linguaggio nasce dalla necessità di offrire ai programmatori uno strumento avanzato, che superi vincoli e convenzioni che i linguaggi più diffusi si portano normalmente dietro. Non c'è da stupirsi per esempio se in Java, utilizzato dal 1995, dobbiamo seguire delle regole che al giorno d'oggi ci sembrano banali, se non superate, ma che magari al tempo avevano una certa rilevanza e/o un certo fondamento. Kotlin si predispone come un passo avanti nella programmazione consentendo però un graduale approccio a tutti gli sviluppatori Java: esso infatti è pensato per compilare sulla Java Virtual Machine, ed è possibile programmare sfruttando classi Java. Essendo Android Studio sviluppato in Java e pensato inizialmente per uno sviluppo in tale linguaggio, Kotlin si adatta benissimo alla programmazione per applicazioni Android, ed è per questo motivo che dal 2017 Google lo ha dichiarato come linguaggio ufficiale per le sue app al pari di Java.

Il seguente report non è pensato come una guida (di cui il web è pieno), bensì una carrellata di peculiarità di questo linguaggio analizzate ed esemplificate sia mediante riferimenti all'applicazione Take the Pill, che abbiamo sviluppato prima in java e poi in Kotlin, sia a semplici pezzi di codice scritti per l'occorrenza. Nei prossimi paragrafi verranno trattati vari aspetti tipici della programmazione, evidenziando analogie e differenze tra i due linguaggi.

SEZIONE 1

VARIABILI, COSTANTI, TIPI DI DATO

1.1 DICHIARAZIONE

La dichiarazione di variabili in Kotlin avviene mediante la keyword '**var**', preceduta dall'indicatore di visibilità e seguita, dopo il nome, dal tipo di dato che si vuole utilizzare.

```
private var day: Int
```

Tuttavia, è possibile omettere la dichiarazione del tipo della variabile, che verrà attribuito automaticamente, se l'istanziazione e la dichiarazione avvengono sulla stessa riga.

```
private var day = 0
```

Nel caso in cui vengano dichiarate delle variabili ed il loro valore non venga mai modificato, la compilazione segnala un 'warning' che avvisa il programmatore di ciò, inducendolo ad ulteriori controlli: se effettivamente il valore è destinato a non cambiare mai, si è invitati a dichiarare una costante; per far ciò è sufficiente inserire la keyword *'**val**'* al posto di *'var'*.
Oltre agli indicatori di visibilità **private**, **protected** e **public** (quest'ultimo assegnato in automatico se non diversamente espresso), è possibile anteporre alcuni modificatori come per esempio *'**lateinit**'*, che assicura al compilatore l'inizializzazione di una variabile non-nullable ma in una riga di codice diversa dalla dichiarazione (senza di essa, verrebbe segnalato errore, come trattato nel prossimo paragrafo).

```
var ora : Time   //errore: la variabile ora potrebbe essere nulla

lateinit var ora : Time   //corretto
```

1.2 GESTIONE VALORI NULL

Uno dei più temuti errori a cui vanno incontro i programmatori Java, il *Null Pointer Exception*, è fortemente contrastato in Kotlin. Già in fase di compilazione infatti viene segnalato un errore in caso di assegnazioni di valori nulli o tentate operazioni su elementi nulli, a meno che non venga esplicitata la volontà di utilizzare un oggetto nullo mediante la sintassi **tipoDato?**. In tal caso l'oggetto si definisce **nullable**.

```
private var nome:String?
private var descrizione:String?
private var tipo:String?


....
this.nome=null
this.descrizione=null
this.tipo=null
```

Ciò porta ad un importante considerazione: quando si dichiara una variabile, e questa viene inizializzata solo dentro ad un metodo della classe (che quindi il compilatore non è certo venga effettivamente chiamato), o si esplicita la possibilità del valore null, o si assicura una successiva corretta inizializzazione mediante la keyword 'lateinit'; in caso contrario verrà segnalato errore.

Il '**?**' come carattere indicante possibili valori nulli viene impiegato anche in fase di controllo, nelle seguenti righe la condizione viene valutata solo se la variabile *th* non è nulla.

```
if(th?.mDays==0){
    Log.d("errore terapia","numero giorni non trovato");
    return null
}
```

E' possibile infine fornire indicazioni su come dev'essere trattata una variabile nel caso fosse nulla:

```
val size=box?.length ?: -1
```

size viene posta uguale a -1 se *box.length*= NULL.

1.3 TIPI DI DATO

In Kotlin vige la seguente regola generale: qualsiasi tipo di dato è considerato oggetto; in fase di esecuzione in realtà alcuni dati che in Java sono primitivi (int, char...) restano tali anche in Kotlin, ma al programmatore appaiono tutti sotto forma di classi, con relative funzioni. Questo consente una maggior libertà d'azione, ma inevitabilmente certe ovvietà del mondo Java non sono più valide: per esempio la conversione da un tipo di dato numerico ad un altro più grande non è automatica, ma avviene solo mediante specifiche funzioni delle relative classi. Inoltre c'è da tenere presente il fatto che l'ammissione di valori nulli determina un vero e proprio tipo di dato, quindi non si può trattare un 'Int' e un 'Int?' allo stesso modo: bisogna effettuare eventualmente un cast mediante la parola chiave **as** (ulteriori approfondimenti nel prossimo paragrafo). Ciò è molto frequente quando si vuole passare come parametro ad un metodo che non accetta valori nulli una variabile che, per nostra comodità, abbiamo precedentemente definito nullable.

```
public class TherapyEntityDB {

    var mID : Int?    //ammette valore nullo

...



val current  = AssumptionEntity(calendar.getTime(),hour,( Th.mID as Int),false)
```

La frequente necessità di convertire in stringa dati numerici, affrontata in Java mediante operazioni di concatenazione, trova in Kotlin un ben più semplice ed efficace strumento: il simbolo '**$**' anteposto alla variabile numerica consente di assegnare tale valore ad un oggetto di tipo stringa. Nel caso si volesse convertire invece un oggetto numerico per passarlo come parametro ad un metodo richiedente stringhe, è sufficiente utilizzare il metodo *toString()*

```
Log.d('n Giorni',giorni.toString())
```

1.4 CAST

Come già dichiarato in precedenza, dichiarando una variabile e inizializzandola in precedenza, è possibile omettere il tipo in quanto il compilatore lo rileva automaticamente; questo è solo un esempio dei frequenti cast impliciti che avvengono durante lo sviluppo in Kotlin.

val word = 'parola a caso' // word viene creata come oggetto String

È tuttavia possibile eseguire un cast esplicito mediante la parola chiave **as** / **as?** (quest'ultima non genera errori se l'oggetto del cast è null)

val number : Long= 5 as Long

Inoltre è molto semplice, data una variabile/costante, leggerne il tipo, mediante la parola chiave **is**

```
if (obj !is String) {
    print ('Not a String')
}
else {
    print(obj.length)
}
```

Come mostrato dall'esempio, questa funzionalità permette di realizzare controlli non solo sul valore ma anche sul tipo stesso degli oggetti trattati.

1.5 CONFRONTO TRA OGGETTI E VARIABILI

Per eseguire il confronto tra due oggetti si ha a disposizione l'operatore '==' (in java è necessario invece invocare il metodo 'equals' o eventuali metodi sovrascritti), mentre per verificare che il contenuto di due variabili di tipo 'primitivo' sia identico, è possibile utilizzare l'operatore '**===**' (in java ciò avviene con '==').

SEZIONE 2

CLASSI

2.1 CREARE UNA NUOVA ISTANZA

In Java, per creare un'istanza di una classe, si utilizza la keyword **new**; questa sintassi è stata ereditata dal linguaggio C++ per mantenere certi livelli di affinità per quanto riguarda la sintassi, in maniera tale da agevolare il programmatore che passa da un linguaggio all'altro. In C++ essa serve a ricordare che, una volta allocata una certa zona di memoria per l'istanza creata, è necessario liberarla, a processo ultimato, mediante la chiamata **delete**, per evitare problemi di memory leak ; in Java tuttavia esiste il garbage collector, la cui funzione è proprio quella di liberare la memoria occupata da oggetti non più utilizzati, rendendo di fatto inutile la chiamata delete ma, di conseguenza, anche la presenza del 'new'.

Anche con Kotlin è presente il garbage collector e si è deciso, per l'inutilità appena descritta unita alla volontà di considerare i costruttori come funzioni normale, di abolire l'uso forzato della keyword sopra trattata. Per istanziare un oggetto è sufficiente quindi invocare l'apposito metodo, senza ulteriori vincoli di sintassi.

2.2 PROPRIETA'

Come in Java, anche in Kotlin si possono assegnare variabili e costanti ad una classe, che ne definiscono la struttura e per tal ragione vengono chiamate 'proprietà'. Nella programmazione è molto frequente l'accesso in lettura e/o scrittura a queste variabili, mediante metodi getter e setter; per agevolare ciò, alcuni ambienti di sviluppo come Android Studio offrono strumenti che generano in modalità automatica questi metodi. In Kotlin si è fatto un ulteriore passo avanti, rendendo implicita la possibilità di leggere e modificare le proprietà di ciascun oggetto (purché non siano private). In fase di sviluppo della classe non è quindi necessario definire molteplici metodi *get* e *set* (ma è comunque possibile, nel caso si voglia personalizzarne il funzionamento) risparmiando tempo e righe di codice; il programmatore che vuole accedere alle proprietà di un'istanza della classe deve semplicemente utilizzare la seguente sintassi:

```
nomeOggetto.proprietàX=‘ valore’;      // equivalente di una chiamata Set
var a=nomeOggetto.proprietàX           // equivalente di una chiamata Get
```

Questa funzionalità ancora una volta rende impossibile (se non forzata) la creazione di oggetti null, proprio per evitare che questi metodi 'di background' possano generare errori.

```
class AddEditTherapyActivity : AppCompatActivity() {

    private var terapia: TherapyEntityDB? = null    // Rappresenta la terapia in considerazione
    lateinit var drugList: Array<String?>/*? = null*/  // Rappresenta la lista dei farmaci
    private var listaOre: ArrayList<IntArray>? = null  // Rappresenta la lista delle ore
    private var giorniSelezionati: BooleanArray? = null    // Usata nella selezione dei giorni
    private val giorni = arrayOf("Lunedì", "Martedì", "Mercoledì", "Giovedì", "Venerdì", "Sabato", "Domenica")
    lateinit var db: DatabaseHelper
    lateinit var tvHours: TextView
    private var day = 0
    private var month = 0
    private var year = 0

    //Codice classe

}
```

2.3 EREDITARIETA' E OVERRIDING

In Kotlin tutte le classi derivano implicitamente da una superclasse chiamata *Any*, similarmente a quanto avviene in Java con la classe *Object*. Tuttavia in *Any* non ci sono altri metodi al di fuori di *equals(), hashCode(), toString().*

Le classi possono derivare anche da altre superclassi, ma ciò è possibile solo se, nella dichiarazione della superclasse, è presente la parola chiave **open.** In caso contrario, ciascuna classe è di default **final,** non ammette cioè classi derivate. Un metodo di una classe open ammette overriding solo se, nella dichiarazione, viene a sua volta firmato come **open**; inoltre, nella classe derivata, si esplicita l'overriding con la parola **override.** Ciascun metodo segnato come override ammette a sua volta l'overriding (se la sua classe è marcata **open**); se si vuole evitare ciò, si utilizza la parola chiave **final.**

```
open class Padre {
    open fun p() {}                  // ammette overriding
    fun pClosed() {}                 // non ammette overriding
}
class Figlio() : Padre() {
   final override fun p() {}         // non ammette ulteriore overriding
}
```

La parola chiave **super** si usa, come in Java, per richiamare una funzione della superclasse.

SEZIONE 3

FUNZIONI

3.1 DICHIARAZIONE

La dichiarazione di una funzione avviene utilizzando la parola riservata **'fun'**, seguita da nome della funzione, eventuali parametri racchiusi tra parentesi tonde (indicando per ognuno il 'NomeTipo' e il 'NomeParametro'), ed infine il tipo di dato restituito preceduto dal simbolo '**:**'.

```
fun removeTherapyByDrug(nome: String?): Int {
    val db = writableDatabase
    val deleteStatus = db.delete(therapyTable,  whereClause: "$therapyDrug=?", arrayOf(nome))
    db.close()
    return deleteStatus
}
```

Le funzioni definite in Kotlin, aggiungono oltre al classico funzionamento di java, la possibilità di poter essere assegnate a variabili, essere passate come argomento ad altre funzioni ed allocarne istanze a runtime. Ciò rimarca ancora una volta la grande versatilità di questo linguaggio e la volontà dei suoi sviluppatori di superare le restrizioni di java non fortemente necessarie.

```
fun circleOperation(radius: Double, op: (Double) -> Double): Double {
    val result = op(radius)
    return result
}
```

Nell'esempio sopra, la funzione denominata "circleOperation" accetta come primo parametro un dato di tipo *Double* e come secondo una funzione. Quest'ultima presenta fra parentesi il tipo di dato accettato (*Double*) e indica con una freccia il tipo di dato restituito (anch'esso di tipo *Double*).

È possibile inoltre, nel caso di una funzione formata da una singola riga, eliminare le parentesi graffe e collocarne il corpo subito dopo la firma della stessa, preceduta dall'operatore ''='.

3.2 VALORI DI INPUT; VALORI DI RITORNO

È possibile (come succede in java), passare ad una funzione un numero illimitato di argomenti (separati da virgola) utilizzando l'operatore '**vararg**', seguito dal nome del parametro e dal tipo di dato.

Per poter fare ritornare due valori ad una qualsiasi funzione è necessario indicare nella firma del metodo, come dato restituito il tipo '**Pair**', seguito dai due tipi di dato (separati da virgola) che si vogliono restituire.

```
fun getUserNameAndState(id: Int): Pair<String?, String?> {
    require( value: id > 0, { "Error: id is less than 0" })

    val userNames: Map<Int, String> = mapOf(101 to "Chike", 102 to "Segun", 104 to "Jane")
    val userStates: Map<Int, String> = mapOf(101 to "Lagos", 102 to "Imo", 104 to "Enugu")

    val userName = userNames[id]
    val userState = userStates[id]
    return Pair(userName, userState)
}
```

È possibile incrementare ulteriormente il numero di valori restituiti sostituendo 'Pair' con il tipo '**Triple**'.

3.3 GESTIONE DELLE ECCEZIONI

In Java esistono eccezioni **Checked** (tutte quelle che estendono la classe *Exception* tranne *RunTimeException*) ed eccezioni **Unchecked** (tutte quelle che estendono *RuntTimeException*). Le prime differiscono dalle seconde per:

- Dichiarazione dell'eccezione nella firma del metodo, mediante parola chiave **Throws**
- Obbligo per il programmatore di gestire le eccezioni ogni qualvolta si usa un metodo che possa lanciarne, mediante il costrutto **Try/Catch**

Il compilatore segna errore nel caso il punto precedente non venga rispettato

L'utilizzo di eccezioni Checked assicura una maggior protezione dagli errori, tuttavia per progetti di notevoli dimensioni ciò influisce pesantemente sulle prestazioni; inoltre costituisce un pesante ed inutile vincolo per le situazioni in cui si ha la certezza matematica che l'errore non possa verificarsi. A fronte di ciò in Kotlin le eccezioni sono sempre Unchecked, permettendo una maggiore versatilità a chi scrive codice. I blocchi Try/Catch sono comunque ammessi ed utilizzati, ma non rappresentano un obbligo e, di conseguenza, il compilatore non genera errore se non utilizzati. Nella firma del metodo non va specificata l'eccezione, che verrà invece lanciata nei punti opportuni con l'espressione **throw**.

3.4 FLUSSI DI CONTROLLO

La gestione dell'ordine con cui vengono eseguite le istruzioni in Kotlin è molto simile a quella di Java, con alcune particolarità. Mentre resta inalterato il funzionamento del ciclo **while** e del blocco **if/else***,* troviamo una funzionalità aggiuntiva nel ciclo **for***;* è semplificata infatti la personalizzazione delle iterazioni, che possono avvenire in altri modi oltre al classico incremento/decremento di un indice, utilizzando delle apposite funzioni messe a disposizione dalle librerie di linguaggio.
Di particolare rilievo infine è il blocco **when:** essa va a sostituirsi alla struttura **switch** utilizzata in Java, presentando una sintassi più snella e un uso leggermente più performante.
Nella forma classica, dopo la keyword 'when', si pone la variabile per cui si vuole controllare i valori, e ,a cascata, i valori per cui, in caso di equivalenza, eseguire l'istruzione o il blocco di istruzioni corrispondente.

```
when (x)    {
0 -> print("X == 0")
1 -> print("X == 1")
else -> {
    print("X non è nè 0 nè 1")
        }
}
```

Nel linguaggio Java, lo stesso codice scritto con uno 'switch' risulta essere meno leggibile.

SEZIONE 4

KOTLIN & ANDROID

4.1 PERCHE' UTILIZZARE KOTLIN NELLO SVILUPPO ANDROID?

- Questo linguaggio è supportato eccellentemente da Android Studio, infatti esso permette di utilizzare il debugging, il completamento automatico, il test per le varie unità e il refactoring;
- Alta interoperabilità: è possibile mescolare nello stesso progetto classi scritte nel linguaggio Java con classi scritte nel linguaggio Kotlin.
  Ciò permette di utilizzare la maggior parte delle librerie e dei framework Java nei progetti Kotlin.

- Lo stesso codice scritto con il linguaggio Java può essere scritto con il linguaggio Kotlin in meno righe e questo comporta ad un codice più pulito (e quindi più leggibile) e ad una dimensione del file finale minore.

  Porzione di codice scritta in linguaggio Java:

```
view.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // Do whatever you want to
}});
```

  Stesso codice scritto in linguaggio Kotlin:

```
view.setOnClickListener { // Do whatever you want to }
```

  Per esempio, per diminuire la quantità di codice, è stata introdotta l'espressione *Lambda* con la quale è possibile definire funzioni anonime (queste espressioni sono state introdotte in Java solamente a partire dalla versione 8). Esse permettono di essere passate direttamente come espressioni e ciò implica il fatto che non è più necessario scrivere le specifiche di una funzione in una classe (o in un'interfaccia astratta).

  Un ulteriore esempio consiste nella creazione di una classe *Singleton,* (ovvero una particolare classe che permette la creazione di una sola istanza all'interno di un programma). Nel linguaggio Java, per far ciò è necessario definire una classe con un costruttore privato, e successivamente creare una tale istanza come attributo privato. In Kotlin tutto questo avviene in una sola riga di codice, ovvero è permessa la creazione di un singolo oggetto, il quale presenta una semantica uguale al Singleton;

- Il linguaggio Kotlin è stato creato in modo da mettere assieme i vantaggi dei *linguaggi funzionali* e dei *linguaggi procedurali*;

- È stata implementata la possibilità, (come succede nel linguaggio C), di aggiungere delle funzionalità ad una classe già esistente (ad esempio l'aggiunta di un metodo). Ciò è possibile creando una funzione di estensione e anteponendo il nome della classe (da modificare) al nome della funzione che si sta creando;
- Con la nuova versione di Kotlin è possibile programmare anche il lato *front-end* di un'applicazione; inoltre è stata implementata l'opportunità di poter scrivere file Gradle. Mediante ciò, è possibile integrare nel progetto numerosi plug-in, medianti i quali l'efficienza e la leggerezza di Kotlin sono ulteriormente incrementate.

4.2 PROSPETTIVE

L'azienda Realm, che negli ultimi anni sta viaggiando a gonfie vele nella produzione di database per dispositivi mobili, ha svolto recentemente una ricerca riguardo l'utilizzo di Kotlin da parte di sviluppatori che operano sulla piattaforma Android. È emerso che, se nel 2016 Kotlin occupava era utilizzato nel 5,1% dei casi rispetto al 94,9% di Java, solamente un anno dopo è arrivato al 14.3%; si registra inoltre che circa il 20% delle app prodotte in Java stanno venendo attualmente riscritte nel nuovo linguaggio, e si prevede che i numeri saranno in costante crescita fino ad arrivare ad un graduale abbandono di Java in questo ambito. Questo trionfo è dovuto sicuramente alla semplicità di Kotlin, alla sua miglior leggibilità, alla facilità di comprensione e, come esemplificato nei paragrafi precedenti e sperimentato da noi stessi nello sviluppo del progetto, al fatto che Kotlin è stato progettato su misura per la programmazione di software mobile.

SEZIONE 5

BIBLIOGRAFIA

https://developer.android.com/kotlin/
https://kotlinlang.org/docs/reference/comparison-to-java.html
https://android-developers.googleblog.com/2018/02/introducing-android-ktx-even-sweeter.html
https://codelabs.developers.google.com/codelabs/taking-advantage-of-kotlin/
https://italiancoders.it/kotlin-cose-cosa-serve-sapere-ai-programmatori-java/
https://kotlinlang.org/docs/reference/typecasts.html
http://www.mokabyte.it/2018/04/kotlin-2/
https://www.theregister.co.uk/2017/10/10/kotlin_killing_java_among_android_devs/
https://code.tutsplus.com/articles/java-vs-kotlin-should-you-be-using-kotlin-for-android-development--cms-27846