

Dipartimento di Ingegneria

dell’Informazione

Corso di Laurea in

Ingegneria Informatica

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

[**1.** **???Titolo Capitolo 1???** 1](#_Toc510638976)

[**Bibliografia** 2](#_Toc510638977)

# 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.

# NULL POINTER EXCEPTION

Uno dei più temuti errori a cui vanno incontro i programmatori Java 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

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.

**UGUAGLIANZA TRA OGGETTI E VARIABILI**

Nel linguaggio kotlin, per poter confrontare due oggetti basta utilizzare l’operatore “==” (in java è necessario invocare il metodo “equals”), mentre per verificare che il contenuto di due variabili sia identico, è possibile utilizzare l’operatore “===” (in java è necessario utilizzare l’operatore “==”).

# FLUSSO DI CONTROLLO: IF, FOR, WHEN, WHILE

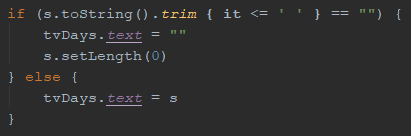
Il flusso di controllo in un linguaggio di programmazione è l’ordine in cui vengono eseguite le istruzioni.

Con le strutture di controllo si definisce quale dev’essere il flusso di controllo delle istruzioni, chiamate a funzione o dichiarazioni nel programma.

Normalmente, le istruzioni sono eseguite in sequenza, una dopo l’altra, all’interno di una funzione o in un metodo, ma tuttavia, in un programma più complesso, si può prevedere un flusso che non sia sempre statico ma dinamico ovvero che cambia in base a determinate circostanze.

* ESPRESSIONE IF

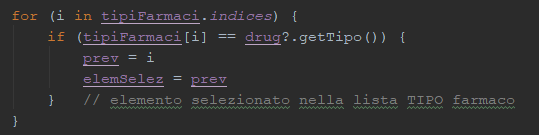
L’espressione if (o if-else), consiste nel verificare che una determinata condizione sia o meno verificata. Se è vera, viene eseguito il blocco posto dopo il controllo; se è falsa, non viene eseguito il blocco, ma se è presente in un caso else, allora verrà eseguito quello.



In generale, l’utilizzo di questa espressione è identico a quella nel linguaggio java.

* CICLO FOR

Il for generalmente è utilizzato o per eseguire un numero prefissate di volte un blocco di istruzioni, oppure per iterare gli elementi di una collezione (ad es. una lista o un array).



In questo caso, il corpo del ciclo “for” viene eseguito un numero di volte pari a “tipifarmaci.indices”.

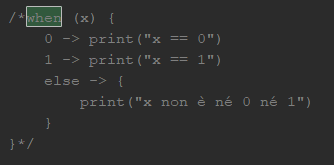
Inoltre, è possibile iterare il ciclo in maniera “personalizzata” (per esempio facendo decrescere l’iterazione o facendola avvenire saltando alcuni passi) utilizzando delle apposite funzioni messe a disposizione dalle librerie di kotlin.

In java questa iterazione viene eseguita utilizzando la classica sintassi “for(inizializzazione; condizione; incremento) {}”

* ESPRESSIONE WHEN

Questa espressione, nel linguaggio Java è comunemente conosciuta come switch, ma in Kotlin, oltre ad avere un nome diverso, presenta una sintassi più “snella” e un uso leggermente più potente.

Nella forma classica, dopo la “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.



Nel linguaggio Java, lo stesso codice risulta essere meno leggibile.

* ESPRESSIONE WHILE

Il ciclo “while” è un costrutto caratterizzato da una condizione e da un blocco di istruzioni eseguito in verità di quella condizione. Quest’ultimo viene ripetuto finché il valore di condizione non diventa falso.

Questo ciclo non presenta differenze sostanziali nel linguaggio Kotlin e Java.

# 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**.

# VARIABILI/COSTANTI

Per dichiarare una variabile utilizzando Kotlin, basterà semplicemente utilizzare la keyword “**var***”* all’inizio della dichiarazione stessa.

private var day: Int

Inoltre, 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 segnalerà un “warning” che ci avviserà di ciò.

Per dichiarare una costante sarà sufficiente inserire la keyword *“***val***”* al posto della keyword *“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).

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

lateinit var ora : Time // corretto

# 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 così anche in Kotlin, ma al programmatore appaiono tutti sotto forma di classi, con relative funzioni. Questo permette 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())

# CAST

In Kotlin avvengono spesso cast impliciti in quanto, se una variabile o una costante viene dichiarata ed inizializzata nello stesso momento, non è necessario specificarne il tipo, che viene impostato in automatico dal compilatore sulla base del valore assegnato.

*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 {*

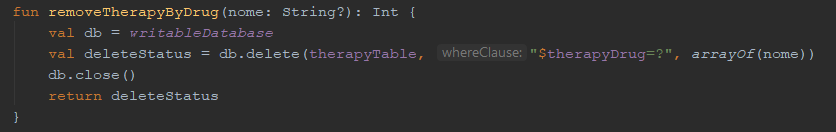
*print(obj.length)*

*}*

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

# FUNZIONI

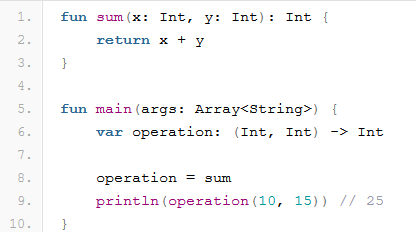
La dichiarazione di una funzione avviene utilizzando la parola riservata **“fun***”*, seguita da nome della funzione, eventuali parametri racchiusi tra parentesi tonde (indicando per ognuno il “NomeTipo” e il “NomeParametro”), ed infine il tipo di dato restituito preceduto dal simbolo “**:**”.



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.

ESEMPIO DA SOSTITUIRE CON UNO NOSTRO

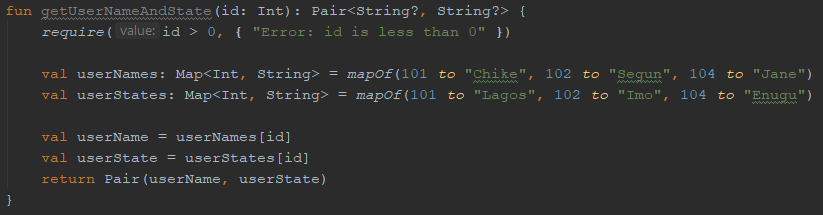
Una variabile di nome “operations” dichiara come tipo una funzione (definita precedentemente con il nome di “sum”), andando ad indicare fra parentesi i parametri accettati (di tipo “int”) e specificando il tipo di dato restituito (indicato con una freccia).



E’ 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 ‘”=”.

* RITORNO DI PIU’ VALORI

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.



Nel caso si volesse far ritornare tre valori, basta sostituire il tipo “Pair” con il tipo “Triple”.

* ARGOMENTI ILLIMITATI

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.

# PROPRIETA’ DELLE CLASSI

Come in Java, 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.

//INSERIRE ESEMPIO NELLA NOSTRA APP

# ISTANZA DI UNA CLASSE

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.

# 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.

**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/

Istruzioni prese dal Sito del prof

“Report: si illustrino le ragioni che hanno spinto i progettisti di Kotlin a sviluppare un nuovo linguaggio; si illustrino le caratteristiche peculiari del linguaggio Kotlin, nonché somiglianze e differenze con il linguaggio Java.”

“Il report deve includere *una bibliografia che elenchi le fonti della ricerca effettuata.  
Il report non deve basarsi su un'unica fonte (ad esempio, la documentazione ufficiale di Google, sebbene tale documentazione sia un buon punto di partenza).  
Non è consentito effettuare operazioni di "taglia e incolla" dalle fonti: il report deve essere una rielaborazione originale delle informazioni trovate, non un riassunto o una copia.  
Poiché gli argomenti sono molto vasti, ciascun gruppo deve eseguire una selezione del materiale da presentare: in altre parole, deve scegliere che taglio dare al report, quali informazioni includere e quali escludere.  
I seguenti aspetti saranno presi in considerazione nella valutazione del report: chiarezza, completezza, capacità di sintesi (il report deve essere completo senza essere prolisso o costellato di dettagli banali), qualità della bibliografia, corrispondenza tra le informazioni selezionate per il report e le informazioni esemplificate nelle app.”*