POLITECNICO DI TORINO
Corso di Laurea Magistrale
in Software e Sistemi Digitali

Tesi di Laurea Magistrale

Studio del linguaggio Kotlin per applicazioni distribuite: Front-End Android

Relatore
Giovanni Malnati

Autore
Stefano Cuscunà

A.A.2017/2018

Dediche

Riconoscimenti

Indice

1 Introduzione **13**
1.1 Linguaggio Kotlin 13
1.2 Stato attuale del linguaggio 13
1.3 Stato dell'arte per le applicazioni distribuite 14
1.4 Scopo della tesi 15
1.5 Organizzazione della tesi 15

2 Kotlin **17**
2.1 Introduzione al linguaggio 17
2.1.1 Storia 17
2.1.2 Caratteristiche 18
2.1.3 Prefazione alla panoramica tecnica 18
2.2 Sintassi di base 19
2.2.1 Package 19
2.2.2 Definizione di variabili 19
2.2.3 Modificatori 20
2.2.4 Espressioni condizionali 20
2.2.5 Gestione dei null 21
2.2.6 Collezioni 22
2.2.7 Eccezioni 23
2.3 Programmazione ad Oggetti 24
2.3.1 Classe 24
2.3.2 Oggetti 27
2.3.3 Ereditarietà 28
2.3.4 Polimorfismo 31
2.4 Programmazione Funzionale 31
2.4.1 Sintassi di base 32
2.4.2 Funzioni Higher-Order 33
2.4.3 Funzione Estensione 34
2.4.4 Funzione Lambda 34
2.4.5 Funzioni Anonime 35
2.4.6 Funzioni Inline 35
2.4.7 Modificatore tailrec 36
2.5 Programmazione Generica 37
2.6 Interoperabilità con Java 38
2.6.1 Utilizzo di Java su Kotlin 38
2.6.2 Utilizzo di Kotlin su Java 41
2.7 Coroutines 44

2.7.1 Funzioni Suspending 45
2.7.2 Interfaccia CoroutineContext 45
2.7.3 Interfaccia Continuation 46
2.7.4 Avviare e fermare una coroutine 46
2.7.5 CompletableFuture 47
2.7.6 Sperimentale 47

3 Sviluppo dell'applicazione **49**
3.1 Scopo dell'applicazione 49
3.2 Requisiti 49
3.2.1 Applicazione Android 49
3.3 Progettazione 50
3.3.1 Richieste HTTP 50
3.3.2 Gradle 51
3.3.3 JSON 51
3.3.4 Applicazione Android 52
3.4 Implementazione 59
3.4.1 Applicazione Android 59
3.4.2 Package Models 60
3.4.3 Package LoginAndRegister 62
3.4.4 Package NearbyPlaces 65
3.4.5 Package OnTheRoad 70

4 Valutazione delle criticità **74**
4.1 Esito dello sviluppo in Kotlin 74
4.2 Kotlin: Vantaggi e Svantaggi 74
4.2.1 Vantaggi 74
4.2.2 Svantaggi 76

5 Conclusioni e Sviluppi Futuri **79**

Elenco delle figure

1.1 Esempio di come funziona una classica applicazione distribuita di tipo Client-Server. 15

3.1 Richiesta Accesso Posizione 54
3.2 Login-Registrazione 55
3.3 Lista/Mappa dei Luoghi Vicini 56
3.4 Descrizione di un luogo 57
3.5 Scelta Itinerario Viaggio 58
3.6 Profilo 58

Listings

2.1 Definizione di un package . . . 19
2.2 Import del package . . . 19
2.3 Dichiarazione variabile . . . 19
2.4 Definizione di una variabile . . . 20
2.5 Sintassi del for . . . 20
2.6 Sintassi del when . . . 20
2.7 Sintassi di when e in . . . 21
2.8 Sintassi di when e is . . . 21
2.9 Variabili nullable . . . 21
2.10 Trattare una variabile nullable come se non lo fosse . . . 22
2.11 Operatore elvis . . . 22
2.12 Uso di collezioni in Kotlin . . . 22
2.13 Eccezioni . . . 23
2.14 Uso di Throw . . . 23
2.15 Esempio senza Nothing . . . 23
2.16 Esempio con Nothing . . . 23
2.17 Sintassi per la dichiarazione di una classe. . . . 24
2.18 Classe innestata . . . 25
2.19 Sintassi del costruttore primario. . . . 25
2.20 Init del costruttore primario . . . 25
2.21 Costruttore secondario . . . 25
2.22 Data class . . . 26
2.23 Classi Sigillate . . . 26
2.24 Companion Object . . . 27
2.25 Istanziazione . . . 27
2.26 Istruzione is . . . 27
2.27 Istruzione as . . . 27
2.28 Istruzione as? . . . 28
2.29 Overloading degli operatori . . . 28
2.30 Classe open . . . 28
2.31 Richiamare il costruttore della superclasse . . . 29
2.32 Override di un metodo . . . 29
2.33 Final . . . 29
2.34 Dichiarazione di un'interfaccia . . . 30
2.35 Implementazione di un'interfaccia . . . 30
2.36 Estensione . . . 30
2.37 Polimorfismo . . . 31
2.38 Sintassi per la dichiarazione di una funzione. . . . 32
2.39 Sintassi alternativa per la dichiarazione di una funzione. . . . 32

2.40 Sintassi alternativa per la dichiarazione di una funzione. 32
2.41 Argomenti nominati. 33
2.42 Higher-Order: passaggio di parametri 33
2.43 Higher-Order: ritorno di una funzione 34
2.44 Funzione estensione: Infix 34
2.45 Funzione Lambda 34
2.46 Uso del parametro it 35
2.47 Uso delle Funzioni Anonime 35
2.48 Funzioni Inline 35
2.49 Esempio tipo di un nodo 36
2.50 Tipo di un nodo con le Funzioni Inline 36
2.51 Funzione non idonea per il tailrec 36
2.52 Funzione idonea per il tailrec 37
2.53 Funzione che usa il modificatore tailrec 37
2.54 Creazione di una classe Generica 37
2.55 Passaggio di un Oggetto qualsiasi 38
2.56 Esempio di una classe in e out 38
2.57 Utilizzo della Classe List di Java 38
2.58 Esempio dell'uso di get e set in Kotlin 39
2.59 Esempio dell'uso di Unit in Kotlin 39
2.60 Esempio dell'uso di spread in Kotlin 40
2.61 Uso del metodo notify 40
2.62 Ritorno di un oggetto della classe Java 40
2.63 Getter e Setter di Java 41
2.64 Dichiarazione di una funzione in Kotlin 41
2.65 Uso della funzione di Kotlin in Java 41
2.66 Dichiarazione di una classe in Kotlin con un campo statico 42
2.67 Uso del campo statico di Kotlin in Java 42
2.68 Dichiarazione di una classe in Kotlin con un campo non statico 42
2.69 Uso del campo non statico di Kotlin in Java 42
2.70 Uso del parametro KClass in Java 43
2.71 Uso dell'annotazione @JvmOverloads 43
2.72 Overload ottenuti da @JvmOverloads 43
2.73 Uso dell'annotazione @Throws 43
2.74 Raw-Type del tipo Nothing 44
2.75 Funzioni Suspending 45
2.76 Prima chiamata di una funzione suspending 45
2.77 Interfaccia CoroutineContext 46
2.78 Interfaccia Continuation 46
2.79 Far partire una coroutine 47
2.80 Sospendere una coroutine 47
3.1 Aggiungere la dipendenza di GSON 51
3.2 Aggiungere le dipendenze di Retrofit 53
3.3 Aggiunta dei permessi nel manifest 54
3.4 Ereditarietà Login-User 60
3.5 Retrofit Interface 61
3.6 Uniconverter Factory 62
3.7 Utilizzo di AsyncTask 63

3.8 Caso di credenziali salvate 64
3.9 Esempio della post per la registrazione 64
3.10 Permesso uso localizzazione 65
3.11 Permesso uso localizzazione2 65
3.12 Ritorno ultima posizione del device 66
3.13 Funzione che richiama l'AsyncTask per la richiesta dei luoghi 66
3.14 Funzione GET e Recycler per la visualizzazione dei luoghi 67
3.15 Custom Adapter per la recycler view e il Filter 67
3.16 Swap Mappa-Lista usando ViewFlipper 69
3.17 Metodo per la visualizzazione della mappa 70
3.18 Metodo nel package Description per richiamare Google Maps 70
3.19 Uso di PlaceAutocompleteFragment 71
3.20 Ordinamento della lista On the Road 71
3.21 Creazione percorso sulla mappa 72
4.1 Tipi Nullable 75
4.2 lateinit e Delegates.notNull 75
4.3 Esempio di codice Java da convertire in Kotlin 76
4.4 Conversione in Kotlin 76
4.5 Conversione in Kotlin Ottimale 76
4.6 Getter in Kotlin 77

Capitolo 1

Introduzione

1.1 Linguaggio Kotlin

Kotlin è un linguaggio open-source, fortemente tipizzato e orientato alla programmazione a oggetti. Il linguaggio è stato sviluppato a partire dal 2011 dall'azienda JetBrains, si basa sulla Java Virtual Machine ed è stato strutturato per interoperare pienamente con la Java Runtime Environment.
L'obiettivo del progetto fin dai suoi albori è stato quello di creare un linguaggio che fosse pienamente compatibile con l'ecosistema del Java, in modo da poter sfruttare l'enorme bagaglio di strumenti e conoscenze già disponibile, superando però le principali criticità storiche del Java.
Kotlin è conciso e, grazie al sistema dei tipi implementato, evita il presentarsi di una serie di errori molto comuni durante la programmazione in Java (come il puntatore a null per citare il più noto).
L'interoperabilità del linguaggio si è spinta sempre più avanti con il passare del tempo e il procedere dello sviluppo, implementando strumenti e compilatori che permettono di utilizzare il Kotlin anche per la programmazione Android e Web.
Al momento della stesura di questa tesi, il linguaggio Kotlin è ancora in sviluppo. Molte funzionalità sono ancora in fase sperimentale.

1.2 Stato attuale del linguaggio

Nel maggio del 2017 Google ha ufficialmente annunciato il suo supporto a Kotlin per lo sviluppo su piattaforma Android. Durante la KotlinConf del 2017, inoltre, JetBrains ha sviluppato la sua applicazione ufficiale relativa all'evento interamente in Kotlin, la quale è

compatibile con dispositivi Android, iOS e web Browser; in questo modo l'azienda ha dimostrato che è possibile utilizzare il linguaggio per sviluppare applicativi per diversi tipi di dispositivi e ambienti. Da questo scenario si evince come Kotlin sia descritto come un codice utilizzabile per diversi tipi di applicativi.
Viene dichiarata inoltre la totale interoperabilità con Java e le sue librerie. Questo comporta la possibilità di migrare a Kotlin tutte le applicazioni attualmente sviluppate in Java; viene inoltre fornito un tool dagli sviluppatori che trasforma il codice Java in codice Kotlin in automatico.
Con il compilatore di Kotlin si può anche generare codice Javascript che può essere eseguito da qualunque browser, introducendo anche la possibilità della programmazione web.
Tramite l'infrastruttura Kotlin/Native è possibile compilare codice Kotlin per ottenere codice nativo. In questo modo si può utilizzare Kotlin per ottenere del codice da poter eseguire in ambienti in cui non è prevista alcuna Virtual Machine, come ad esempio iOS. Kotlin/Native, però, è sperimentale e quindi potrebbero ancora esserci problemi che verranno successivamente risolti.

1.3 Stato dell'arte per le applicazioni distribuite

Un'applicazione distribuita è un software che consiste in due o più processi che comunicano tra di loro tramite un canale condiviso. Non è obbligatorio che i diversi processi si trovino su macchine diverse, ma è palese che il principale vantaggio di sviluppare software distribuito è quello di poter utilizzare più device per raggiungere un unico scopo. Nella grande maggioranza dei casi il pattern utilizzato in questo tipo di applicazioni è di tipo client-server. Un dispositivo client invia, tramite il canale di comunicazione, una richiesta di un certo tipo e il server elabora la richiesta e fornisce una risposta.
Si potrebbe verificare, anche, la probabilità che ogni device giri in un sistema differente e il software sarà scritto con un linguaggio diverso. In questo contesto, quindi, per lo sviluppo di applicativi è necessario padroneggiare diverse competenze e conoscere più linguaggi per poter sviluppare tutte le porzioni necessarie al funzionamento del software.

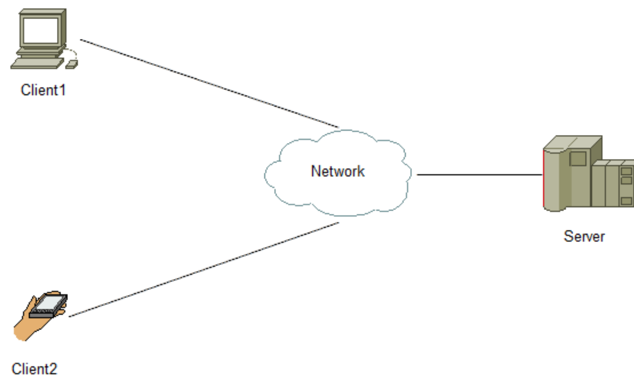

Figura 1.1: Esempio di come funziona una classica applicazione distribuita di tipo Client-Server.

1.4 Scopo della tesi

Lo scopo di questa tesi è lo studio del linguaggio Kotlin nelle sue funzionalità attuali e potenziali al fine di verificare la possibilità di utilizzare Kotlin per lo sviluppo di un'applicazione Android facente parte di un'applicazione distribuita. Per estensione quindi si vuole dimostrare se l'utilizzo di Kotlin per lo sviluppo di applicazioni Android è attualmente possibile e se in futuro può essere considerato come una valida alternativa al Java.
Ad una fase iniziale di studio del linguaggio e degli strumenti utilizzabili, segue una fase di sviluppo in cui verrà realizzata un'applicazione Android di esempio. Lo sviluppo è finalizzato a verificare in un ambito realistico l'effettiva fattibilità delle soluzioni studiate in precedenza.
Dopo il lavoro svolto dovrebbe essere possibile avere una visione chiara dello stato attuale del linguaggio e dove si deve agire per poter raggiungere lo scopo di avere Kotlin come punto di riferimento principale nello sviluppo delle applicazioni Android future.

1.5 Organizzazione della tesi

Per rappresentare al meglio lo studio del linguaggio di programmazione, la tesi sarà suddivisa come segue:

- **Storia e studio del linguaggio**: dalla sua nascita allo studio di tutte le funzionalità che esso propone

- **Sviluppo dell'applicazione**: requisiti, progettazione e implementazione dell'applicativo

- **Valutazioni di criticità**: elenco dei pro e dei contro riscontrati con lo sviluppo del progetto
- **Conclusioni**: mettendo in evidenza i risultati raggiunti

Capitolo 2

Kotlin

2.1 Introduzione al linguaggio

2.1.1 Storia

Nei primi anni 2000 la JetBrains era un'azienda conosciuta nel mondo informatico soprattutto per il loro IDE **IntelliJ IDEA** sviluppato in Java. I limiti di Java erano però piuttosto evidenti e si delineava sempre più la necessità di passare a un nuovo linguaggio, ma senza dover perdere tutti gli strumenti scritti in Java che avevano già sviluppato. Un altro requisito era la compilazione statica, in modo tale da poter superare i problemi relativi ai tipi che affliggevano il Java.
Valutarono quindi diversi linguaggi compatibili con la JVM, tra cui uno dei principali candidati era Scala. Dopo essersi scontrati con la lentezza di compilazione e con la difficoltà intrinseca del linguaggio, però, decisero di abbandonare quella via.
Nel 2011 JetBrains iniziò il progetto per sviluppare un proprio linguaggio di programmazione che vide la prima versione stabile nel 2016 con Kotlin 1.0. Il programma è stabile e molto utilizzato anche da attori importanti del mercato informatico: Gradle, Uber, Spring, Evernote, Corda e molti altri.
L'ambito in cui è utilizzato maggiormente è il mondo Android dove sta ottenendo un consenso importante. Il linguaggio è ufficilmente presente nell'IDE Android Studio 3.0 e Google stessa ha annunciato il supporto al linguaggio per Android durante il Google I/O del 2017.
Lo sviluppo di Kotlin è continuato oltre il rilascio della prima versione e sono state aggiunte funzioni ulteriori e molte sono in fase di sperimentazione.

2.1.2 Caratteristiche

Kotlin è un linguaggio flessibile che può essere utilizzato sia per la programmazione a oggetti che per quella funzionale. Prende apertamente ispirazione da Java, C#, Scala e Groovy.
La funzione principale del linguaggio e una delle principali ragioni del suo successo è sicuramente la piena compatibilità con la JVM. Kotlin permette inoltre di utilizzare tutte le librerie già sviluppate per Java e questo dà accesso alla più grande raccolta di funzioni e framework in ambito informatico. L'idea base è quella di poter convertire il più facilmente possibile qualunque progetto Java in Kotlin; viene anche fornito un tool che riconosce il codice Java e lo trasforma automaticamente in Kotlin; non è uno strumento infallibile, ma permette un veloce passaggio tra i due linguaggi con un intervento minimo da parte del programmatore.
Kotlin è completamente open-source; il suo codice sorgente si trova su gitHub e può essere utilizzato da chiunque in modo totalmente gratuito.
Il linguaggio è molto conciso; è stato calcolato che in media viene scritto il 40% di codice in meno rispetto a Java. Grazie alla sua struttura rigida dei tipi, impedisce il presentarsi di un'intera classe di errori come il *NullPointerExcepiton*.

2.1.3 Prefazione alla panoramica tecnica

Nelle sezioni seguenti di questo capitolo verrà fornita un'esegesi sul linguaggio Kotlin e sulle sue carattestiche, scendendo nel dettaglio delle varie funzionalità del codice. In questo modo sarà possibile capire le parti successive dedicate alla spiegazione dello sviluppo vero e proprio.
La seguente panoramica tecnica verrà esposta con un taglio diverso dalla semplice spiegazione di un nuovo linguaggio. L'obiettivo del presente elaborato è quello di studiare se e come sia possibile migrare a Kotlin dai linguaggi attualmente utilizzati; verrà quindi spiegato il linguaggio nell'ottica di differenza nelle caratteristiche fondamentali della programmazione rispetto ad altri linguaggi. Dopo aver spiegato in cosa è diverso, allora verranno esposte anche l'interoperabilità con Java e le funzioni esclusive di Kotlin.

2.2 Sintassi di base

In questo paragrafo verrà affrontata un breve resoconto della sintassi di base del linguaggio; in modo da affrontare le seguenti sezioni con le conoscenze utili a comprenderne il significato.

2.2.1 Package

In informatica il package è lo spazio dei nomi a cui appartengono gli elementi che compongono il programma. Il meccanismo ha lo scopo principale di riunire gli elementi logicamente correlati in un unico spazio dei nomi. In Kotlin la definizione del package avviene nel seguente modo:

Listing 2.1: Definizione di un package

```
1 package nomePackage
```

Quando bisogna utilizzare un elemento che appartiene a un package differente da quello in cui si sta lanciando l'istruzione, si deve definire anche il package prima dell'elemento. In alternativa è possibile importare lo spazio dei nomi all'inizio per poterlo utilizzare senza scriverlo ogni volta:

Listing 2.2: Import del package

```
1 import nomePackage
```

Tutti gli elementi in Kotlin devono appartenere a uno spazio dei nomi.

2.2.2 Definizione di variabili

In Kotlin ci sono due tipi di variabili che possono essere definite:

- **val**: costanti che possono essere assegnate una volta e poi non possono essere modificate
- **var**: varibili che possono essere modificate tutte le volte che si vuole

Essendo Kotlin un linguaggio fortemente tipizzato, ogni variabile ha un tipo ben definito:

Listing 2.3: Dichiarazione variabile

```
1 var nomeVariabile: tipoVariabile
```

Il tipo può essere ricavato direttamente dall'operazione di assegnazione:

Listing 2.4: Definizione di una variabile

```
val nomeVariabile = Integer(2)
```

In questo caso il tipo di Classe è ben definito e quindi lo stesso compilatore si occupa di assegnare il tipo alla variabile.

2.2.3 Modificatori

Ogni variabile può essere associata a dei modificatori di visibilità che definiscono lo scope di quella variabile:

- **private**: visibile solo all'interno del file che contiene la dichiarazione
- **internal**: visibile solo all'interno dello stesso modulo
- **protected**: non è visibile nelle dichiarazioni di livello superiore
- **public**: visibile a tutti

2.2.4 Espressioni condizionali

Spiegare tutte le espressioni condizionali è superfluo, perché hanno la stessa sintassi della maggior parte dei linguaggi di programmazione comuni. I casi in cui sono diversi sono spiegati di seguito.
Il *for* è usato prevalentemente per scorrere gli oggetti che prevedono un iteratore. La sintassi è la seguente:

Listing 2.5: Sintassi del for

```
for (item: Int in Collection) {
    // codice ciclo
}
```

In questo modo scorre tutti gli elementi dell'oggetto Collection e in ogni ciclo il singolo elemento estratto si trova nella variabile item e può essere manipolato attraverso di essa.
L'istruzione *when* è utile quando bisogna fare determinate azioni in base al valore di una variabile. La sintassi è la seguente:

Listing 2.6: Sintassi del when

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

Dipendentemente dal valore di x, verrà eseguita l'istruzione relativa; mentre, *else* è usato per definire il caso in cui il valore di x non è stato considerato nelle condizioni precedenti.
Sono presenti anche delle parole chiave utili per una più ampia definizione delle condizioni:

- **in**: Definisce la condizione di presenza o meno in una collection
- **is**: Definisce la condizione di appartenenza a una determinata classe

Listing 2.7: Sintassi di when e in

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

Listing 2.8: Sintassi di when e is

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

2.2.5 Gestione dei null

In Kotlin, così come nella maggior parte degli altri linguaggi, la parola chiave null è utilizzata per indicare un oggetto o una variabile che non puntano a un dato consistente. Il sistema dei tipi di Kotlin impone che i tipi che possono essere null vengano dichiarati a tempo di compilazione.

Listing 2.9: Variabili nullable

```
var x : String //variabile non nullable
var x : String? //variabile che puo' essere nullable
```

Se una variabili viene dichiarata nullable, allora verrà trattata normalmente e il programmatore si assume la responsabilità di gestirla. Se viene dichiarata non nullable, allora il compilatore darà errore se vengono scritte istruzioni che rischiano di rendere la variabile *null*. Questo previene il lancio di *NullPointerException*, perché il compilatore stesso ne segnala il rischio. Ci sono casi in cui è necessario utilizzare una variabile nullable come se non lo fosse, ad esempio quando deve essere chiamata una funzione su una determinata variabile.

Listing 2.10: Trattare una variabile nullable come se non lo fosse

```
x!!. length
x?. length
```

Utilizzando *!!* si impone al compilatore di accettare l'istruzione; se la variabile è *null* viene lanciata l'eccezione.Viene usata solo se si è totalmente certi che in questo specifico caso la variabile è valida. Nel caso di *?* l'istruzione viene eseguita solo se la variabile non è *null*, altrimenti viene ignorata.
Molto utile in questo caso si rivela l'operatore elvis: *?:*

Listing 2.11: Operatore elvis

```
val l = b?.length ?: −1
```

In questo caso nella variabile *l* viene salvato il valore di b.length se b non è *null*, altrimenti -1.

2.2.6 Collezioni

Kotlin differenzia le collezioni in due tipi: mutabili e non mutabili.

- *mutabili*: per il cambiamento e quindi la modifica della collezione (aggiunta/rimozione di un elemento etc...).
- *non mutabili*: usate solo per operazioni di lettura (size, get etc..).

Ad ogni tipo di collezione mutabile corrisponde la sua collezione non mutabile: List -> MutableList, Set -> MutableSet, Map -> MutableMap.
Come in Java, appartengono tutte alla classe Collection.

Listing 2.12: Uso di collezioni in Kotlin

```
val number: MutableList<Int> = mutableListOf(1,2,3)
val readNumber: List<Int> = number
print(readNumber) //[1,2,3]
number.add(4)
print(readNumber) //[1,2,3,4]
//readNumber.clear() Non permette la compilazione
number.clear()
print(readNumber) //[]
```

Come si può evincere dall'esempio, a *readNumber* non è permesso usare il metodo *clear()* per cancellare tutti gli elementi dalla lista, poiché esso è stato dichiarato come una collezione di tipo non mutabile.

2.2.7 Eccezioni

Appartenenti alla classe Throwable, sono costituite da uno stack trace (sequenza dei metodi che erano stati eseguita dall'ultimo all'indietro) e da una causa opzionale.
Per catturare una eccezione si usa l'espressione try/catch.

Listing 2.13: Eccezioni

```
try
{
    //corpo try
}
catch(e: Exception)
{
    //corpo catch
}
 finally
{
    //corpo finally
}
```

Finally: si usa opzionalmente per prevedere qualcosa sempre e comunque dopo il try/catch. Viene eseguita in ogni caso (sia che il try venga eseguito senza che abbia sollevato eccezione, sia che il catch abbia catturato l'eccezione).
Per sollevare un'eccezione, si può usare la clausola *throw*:

Listing 2.14: Uso di Throw

```
val n:Double = 10.0
if(n==0.0)
    throw Exception("Eccezione")
print(1/n)
```

Si possono gestire più eccezioni in cascata, rispettando la gerarchia (da quella più generica a quella più specifica).

Tipo Nothing

E' il tipo speciale della espressione throw. Esso non ha valori di ritorno; tipicamente viene utilizzato per contrassegnare quelle parti di codice che potrebbero sollevare un'eccezione:

Listing 2.15: Esempio senza Nothing

```
fun main(arg:Array<String>)
{
    val person = People()
    val s = person.name ?: throw IllegalArgumentExecption("Inserisci il nome")
    println(s)
}
```

Listing 2.16: Esempio con Nothing

```
fun main(arg:Array<String>)
{
```

```
    val person = People()
    val s = person.name ?: fail("Inserisci il nome")
    println(s)
}

fun fail(messaggio:String) : Nothing
{
    throw IllegalArgumentExecption(message)
}
```

Questi due esempi mostrano due modi speculari nel gestire l'eccezione: sollevandola direttamente oppure usando il tipo Nothing.

2.3 Programmazione ad Oggetti

La programmazione a oggetti è un paradigma nato negli anni '70 e diventato il più utilizzato a partire dagli anni '90. Un linguaggio implementa questo paradigma se permette la definizione di oggetti, intesi come unione di strutture e metodi, che interagiscono tra di loro. Ogni oggetto quindi contiene un insieme di attributi che ne definiscono lo stato e una serie di metodi che definiscono come avviene l'interazione con altri oggetti. La programmazione a oggetti si basa su alcuni concetti fondamentali: classe, oggetti, ereditarietà e polimorfismo.

2.3.1 Classe

Una classe è la struttura di base che viene usata come stampo dalla quale successivamente vengono creati gli oggetti.
Le classi in Kotlin vengono dichiarate nel seguente modo:

Listing 2.17: Sintassi per la dichiarazione di una classe.

```
class nomeClasse
{
    private var nomeProprieta: int

    public fun nomeMetodo(val nomeAttributo): valoreRitorno
    {
        //Codice metodo
    }
}
```

All'interno di una classe possono essere presenti variabili (proprietà) e funzioni (metodi). Le proprietà definiscono lo stato dell'oggetto appartenente alla classe; i metodi definiscono come elementi esterni interagiscono con l'oggetto.

Classi innestate

Una classe è detta innestata se è dichiarata all'interno del body di un'altra classe.

Listing 2.18: Classe innestata

```
class Esterna {
    class Innestata {
    }
}
```

Se dichiarata con la parola chiave *inner*, una classe innestata può utilizzare le proprietà e i metodi della classe esterna.

Costruttori

Ogni classe può definire dei metodi particolari che si chiamano costruttori. Il costruttore implementa il codice che viene utilizzato al momento in cui un oggetto appartenente alla classe viene creato. In Kotlin possono essere definiti costruttori primari e secondari. Il costruttore primario è uno solo ed è chiamato ogni volta che viene creato un oggetto.

Listing 2.19: Sintassi del costruttore primario.

```
class NomeClasse constructor(nomeVar : String)
{
    //codice della classe
}
```

In questo caso il costruttore primario non contiene codice, ma viene generato in automatico dal compilatore in base ai parametri passati. Se si vuole scrivere del codice relativo al costruttore primario, si deve utilizzare la parola chiave *init*

Listing 2.20: Init del costruttore primario

```
class NomeClasse(name: String) {
    val prop = name

    init {
        println(prop)
    }
}
```

I costruttori secondari possono essere più di uno e vengono dichiarati utilizzando la parola chiave *constructor*:

Listing 2.21: Costruttore secondario

```
class NomeClasse (val name: String){
    val prop = name

    constructor (name: String) : this(name) {
        //codice del costruttore
```

```
    }
}
```

Ogni costruttore secondario deve richiamare quello primario tramite la parola chiave *this*.

Data Class

Spesso nella programmazione a oggetti vengono create delle classi che hanno il solo compito di contenere e rendere disponibili dei dati.
In Kotlin le *data class* hanno esclusivamente questo scopo.

Listing 2.22: Data class

```
data class User(val name: String, val age: Int)
```

Con questo tipo di classi il compilatore crea automaticamente le seguenti funzioni: equals(), hashCode(), toString(), le componentN() e copy().
Una classe di dati per essere consistente deve rispettare i seguenti requisiti: il costruttore primario deve avere almeno un parametro; tutti i parametri devono essere marcati come val o var; si possono implementare solo interfacce; non possono essere astratte, aperte, innestate o sigillate.
Le funzioni componentN() servono a selezionare i componenti della classe (oggetto.component1() seleziona il primo componente). Se un supertipo ha una funzione componentN() open, essa viene sovrascritta e se non è possibile farlo viene riportato un errore.
Il compilatore crea in automatico la classe, i costruttori, le proprietà e relativi getter e setter.

Classi sigillate

Le classi sigillate (sealed) rappresentano una gerarchia di classi soggetta a restrizioni. Il concetto è molto simile alla classe Enum, ma riguarda i tipi che può assumere un oggetto che appartiene alla classe *sealed*.

Listing 2.23: Classi Sigillate

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1:Expr, val e2:Expr) : Expr()
object NotANumber: Expr()
```

Questo definisce che un oggetto di tipo *Expr* non può essere istanza di altre classi oltre a *Const*, *Sum* e *NotANumber*.

Companion object

In Kotlin le classi non hanno metodi statici. Tuttavia è comunque possibile richiamare un metodo senza dover necessariamente istanziare la classe, utilizzando il costrutto *companion object*.

Listing 2.24: Companion Object

```
class Clasee {
    companion object factory {
        fun create(): Classe = Classe()
    }
}
```

In questo modo la funzione *create* può essere chiamata senza instanziare la classe.

2.3.2 Oggetti

Un oggetto è una locazione di memoria dedicata a un elemento unico e definito che è stato istanziato da una classe. L'oggetto in questione ha le stesse caratteristiche della classe da cui deriva.
In Kotlin si istanzia un oggetto con la seguente istruzione:

Listing 2.25: Istanziazione

```
val oggetto = Classe()
```

Non è necessaria la parola chiave *new*.

Controllo dei tipi e cast

Essendo Kotlin un linguaggio fortemente tipizzato, potrebbe essere necessario verificare da quale classe è stato istanziato un oggetto. Con l'utilizzo della parola chiave *is* è possibile verificare a runtime a quale tipo appartiene un oggetto.

Listing 2.26: Istruzione is

```
if(x is String)
{
    println(x.lenght) //cast automatico
}
```

In questo caso se la variabile x è un'istanza di String, la condizione è verificata. Inoltre il compilatore fa un cast automatico all'interno dell'if e quindi può essere usata come String senza ulteriori istruzioni.
Se il cast non è possibile, viene generata un'eccezione.
In caso di cast non sicuro si usa la parola chiave *as*.

Listing 2.27: Istruzione as

```
val x: String = y as String
```

Se la variabile è *null*, si può usare la parola chiave *as?*

Listing 2.28: Istruzione as?

```
val x: String? = y as? String?
```

Overloading degli operatori

Potrebbe essere necessario modificare le funzioni degli operatori per un determinato oggetto, in modo da specificare un comportamento particolare che quest'ultimo deve avere quando gli operatori vengono richiamati.
Per fare ciò si utilizza la parola chiave *operator*.

Listing 2.29: Overloading degli operatori

```
class Classe(val number: Int)
{
    operator fun plus(increment: Int) : Int
    {
        return (number + increment)
    }
}
```

In questo modo quando viene chiamato l'operatore + su un oggetto di tipo *Classe*, l'operazione ritorna la somma di *number* più il numero a cui viene sommato.

2.3.3 Ereditarietà

Nella programmazione a oggetti l'ereditarietà è una relazione stabilita tra due classi. Una classe che ne eredita un'altra è detta sua sottoclasse. La superclasse definisce dei metodi e degli attributi che la sottoclasse deve implementare.
In Kotlin una classe può essere ereditata solo nel caso in cui viene definita con la parola chiave *open*.

Listing 2.30: Classe open

```
open class Base(p: Int)

class Derivata(p: Int) : Base(p)
```

Nel caso in cui la classe base ha un costruttore primario, i tipi base devono essere passati anche sulla classe derivata in modo da inizializzarli. Nel caso siano presenti costruttori secondari che devono essere

richiamati nella classe derivata è possibile utilizzare la parola chiave *super*.

Listing 2.31: Richiamare il costruttore della superclasse

```
open class Base(p: Int)
{
    constructor(string : String){}
}

class Derivata(p: Int) : Base(p)
{
    constructor(string : Stringa) : super(string)
}
```

super può essere utilizzata anche per richiamare i metodi e le proprietà appartenenti alla superclasse. Tutte le classi di Kotlin hanno come superclasse comune *Any*.

Override

L'override è l'atto di sovrascrivere un dato elemento. Nell'ereditarietà è possibile per una sottoclasse fare l'override di un metodo che deve avere un'implementazione diversa rispetto alla classe da cui viene ereditato. Anche i metodi devono avere la parola chiave *open* per poter essere sovrascritti.

Listing 2.32: Override di un metodo

```
open class Base {
    open fun v() {}
    fun nv() {}
}
class Derivata() : Base() {
    override fun v() {}
}
```

Un metodo sovrascritto è esso stesso considerato *open*; per impedire un'ulteriore override si utilizza la parola chiave *final*.

Listing 2.33: Final

```
open class AltraDerivata() : Base() {
    final override fun v() {}
}
```

Classi astratte

Prima di poter parlare di classi astratte bisogna introdurre il concetto di metodo o proprietà astratta; sono degli elementi non implementati, ma solo dichiarati. Questo comporta che un metodo astratto non può essere utilizzato fino al momento in cui viene implementato.

Una classe astratta è una classe che contiene almeno un metodo astratto e che per questa ragione non può essere istanziata. Viene dichiarata premettendo la parola chiave *abstract* alla sua dichiarazione. Questa classe serve solo per essere ereditata da altre classi che possono implementare i metodi astratti.

Interfacce

In Kotlin non esiste l'ereditarietà multipla, cioè non è possibile avere una classe che ne estende più di una. Esattamente come nel Java, quindi, viene utilizzato il concetto di interfaccia per questa ragione.
Le interfacce possono contenere metodi astratti, ma non possono salvare uno stato; quindi le proprietà di un'interfaccia devono essere tutte astratte. I metodi devono essere tutti open, ma non è obbligatorio che ci siano metodi astratti.
Per dichiarare un'interfaccia:

Listing 2.34: Dichiarazione di un'interfaccia

```
interface Interfaccia {
    fun bar()
    fun foo() {
      // corpo funzione
    }
}
```

Per implementare un'interfaccia in una classe si utilizzano le seguenti istruzioni:

Listing 2.35: Implementazione di un'interfaccia

```
class Child : Interfaccia {
    override fun bar() {
        // corpo funzione
    }
}
```

Le stesse interfacce ne possono ereditare altre.

Estensioni

In Kotlin è possibile estendere delle classi con nuove funzionalità senza utilizzare l'ereditarietà.

Listing 2.36: Estensione

```
fun Class().newMethod()
{
    //Implementazione del nuovo metodo
}
```

In questo modo la classe viene estesa con il nuovo metodo senza la necessità di creare una classe figlia. L'estensione non modifica la classe già esistente, ma semplicemente rende chiamabile il metodo appena implementato tramite la notazione puntata sulla classe.
L'estensione viene risolta staticamente e non viene fatta alcuna valutazione a tempo di esecuzione.
Se viene dichiarata una funzione di estensione per una classe che ha lo stesso nome e gli stessi parametri di una sua funzione membro, quest'ultima ha la precedenza nelle chiamate successive.

2.3.4 Polimorfismo

Nella programmazione a oggetti il polimorfismo è la proprietà degli oggetti di appartenere contemporaneamente a più classi derivate le une dalle altre. Se un oggetto è istanziato da una certa classe A, esso può essere considerato appartenente anche a una classe B da cui A deriva.

Listing 2.37: Polimorfismo

```
1  open class Base (var n : Int)
2  {
3      open fun sum (x : Int) = x + n
4  }
5
6  class Derived (var m : Int) : Base(n=10)
7  {
8      override fun sum (x : Int) = x + n + m
9  }
10
11 fun main (args : Array<String>)
12 {
13     val base : Base = Derived(5)
14
15     println (base.sum(5) //stampa 20
16 }
```

In questo esempio viene rappresentato come avvengono le chiamate polimorfiche ai metodi. Anche se l'oggetto è di tipo Base, la VM riconosce che è stato istanziato come Derived e si richiama il metodo di quest'ultimo.

2.4 Programmazione Funzionale

Il concetto di programmazione funzionale risiede nell'uso delle funzioni, da un punto di vista matematico del termine.
Rispetto alla programmazione imperativa, con la quale siamo noi a costruire il risultato, la programmazione funzionale tende ad usare un

approccio di tipo dichiarativo: in questo caso l'attenzione è concentrata su come ottenere il risultato.
L'obiettivo di tale programmazione è quello di avere un codice conciso, senza utilizzo di mutazioni di variabili esplicite, portando così ad una migliore correttezza nel codice e ad una conseguente diminuzione di bug.

2.4.1 Sintassi di base

Sintassi per la dichiarazione di una funzione:

Listing 2.38: Sintassi per la dichiarazione di una funzione.

```
1 fun funcName(valName: valueType) : valueType
2 {
3   return valName
4 }
```

La funzione viene dichiarata attraverso la parola chiave *fun*, seguita dal nome della funzione.
All'interno delle parentesi tonde, saranno dichiarati i parametri della funzione, specificando il tipo di valore subito dopo i due punti.
Il valore di ritorno della funzione sarà specificato dopo la chiusura della parentesi tonda, subito dopo i due punti (nel caso in cui venga implementata una funziona che non necessita valore di ritorno, esso sarà dichiarato con la parola chiave *Unit* oppure omesso).
Ecco altri modi per la dichiarazione di una funzione:

Listing 2.39: Sintassi alternativa per la dichiarazione di una funzione.

```
1 fun funcName(valName: valueType)= valName
```

Listing 2.40: Sintassi alternativa per la dichiarazione di una funzione.

```
1 fun funcName(valName: valueType = value)= valueName
```

Nel secondo caso si può vedere come assegnare un valore di default(value), nel caso in cui non venga assegnato quando la funzione viene richiamata; è preferibile che i parametri con valore di default vengono dichiarati alla fine della funzione, in modo tale che, quando essa verrà richiamata, saranno inseriti solo i valori dei parametri che non hanno valore di default. Nel caso dell'override di un metodo, il valore di default deve essere omesso.
Antecedentemente dalla dichiarazione della funzione, come in altri linguaggi, può essere specificato il tipo di visibilità che la funzione può assumere:

- *private*: visibile solo all'interno della classe stessa
- *protected*: visibile anche alle sottoclassi
- *internal*: visibile all'interno del modulo (progetto)
- *public*: visibile a tutti

Mentre, nel momento in cui si ha la necessità di utilizzare una funzione implementata, essa può anche essere richiamata sfruttando gli argomenti nominati:

Listing 2.41: Argomenti nominati.

```
funcName(valName=value)
```

In questo modo si assegna in modo diretto ed esplicito il valore *value* che il parametro *valName* deve assumere.
Ciò permette di passare gli argomenti della funzione in modo differente rispetto a come essa li ha dichiarati (utile nel caso in cui si vogliano sfruttare i valori di default).
Kotlin supporta diversi tipi di funzione:

- Funzioni locali: dichiarate all'interno di altre funzioni
- Funzioni membro: dichiarate all'interno di una classe
- Funzioni ad alto livello: dichiarate all'esterno della dichiarazione di una classe

2.4.2 Funzioni Higher-Order

Nel caso in cui si voglia passare una funzione come argomento di un'altra, o si ha la necessità di implementare un metodo che ritorni una funzione, vengono usate le funzioni higher-order.

Listing 2.42: Higher-Order: passaggio di parametri

```
fun nomeFunzione (funzioneInterna : (param1, param2) -> valoreRitorno)
{
    // corpo funzione
}
```

In questo esempio viene illustrata la dichiarazione di una funzione (*nomeFunzione*), che passerà come parametro un'altra funzione (*funzioneInterna*). La funzione *funzioneInterna* viene dichiarata usando sempre la sintassi di base, illustrata precedentemente nel Listing 2.38.

Listing 2.43: Higher-Order: ritorno di una funzione

```
fun nomeFunzione (param1: Tipo):(param) -> valoreRitorno
{
    // corpo funzione
}
```

In questo caso, invece, *nomeFunzione* tornerà una funzione con parametro *param* e valore di ritorno *valoreRitorno*.

2.4.3 Funzione Estensione

Nel momento in cui, si voglia aggiungere una funzione ad una classe senza fare un override, viene utilizzata la parola chiave *infix*.

Listing 2.44: Funzione estensione: Infix

```
fun nomeClasse.estensione (param: Tipo) : valoreRitorno
{
    // corpo funzione
}
```

In questo modo, è possibile richiamare, da un oggetto appartenente alla classe *nomeClasse*, il metodo *estensione*, come se fosse un metodo di quella classe.

2.4.4 Funzione Lambda

Una funzione lambda non è altro che una funzione non dichiarata, ma passata direttamente come parametro. Il suo utilizzo necessita, nel caso in cui si ha di bisogno, di implementare un blocco di codice da passare agli algoritmi o alle funzioni asincrone. Rispetto alla dichiarazione delle funzioni viste fino ad ora, essa non ha un nome, viene definita al momento della chiamata.
La sua efficacia si perde nei casi in cui l'azione che si vuole specificare non è né comune, né semplice: in questi frangenti è consigliabile implementare una funzione vera e propria.

Listing 2.45: Funzione Lambda

```
val sum = { x: Int, y: Int -> x + y }
```

In questo esempio viene creata una funzione lamba per la somma di due numeri; il risultato verrà memorizzato su *sum*. Prima di -> si trovano i parametri da passare alla funzione e dopo, invece, va scritto il corpo della funzione.

Parametro it

Nella maggior parte dei casi le funzioni lambda hanno un solo parametro in ingresso. In questo caso può essere usato *it* come nome del parametro.

Listing 2.46: Uso del parametro it

```
fun nomeFunzione(param: Int, funzione:(Int)->Boolean)
{
    //corpo funzione
}

fun main(args : Array<String>)
{
    nomeFunzione(1, {it>0})
}
```

In questa funzione lambda viene passato un solo parametro; per tale ragione può essere omessa la definizione dei parametri e passare solo il body utilizzando *it*.

2.4.5 Funzioni Anonime

Nella funzione lambda non può essere specificato il tipo di ritorno. Tipicamente questo non è un problema perché il tipo può essere facilmente dedotto. Mentre, nel caso in cui questo non sia possibile, vengono in aiuto le funzioni anonime:

Listing 2.47: Uso delle Funzioni Anonime

```
print(fun(par:Int):Int{return par+10})
```

La dichiarazione è identica a quella di una funzione normale, tranne che non si ha la necessità di mettere il nome.

2.4.6 Funzioni Inline

Sono delle funzioni che non vengono chiamate. Quando c'è un riferimento a una funzione inline il compilatore sostituisce la chiamata con il codice stesso della funzione. In altre parole viene sostituito il riferimento con il corpo stesso della funzione.
Lo svantaggio nell'uso delle funzioni inline è la crescita del codice generato in compilazione; mentre il suo vantaggio è quello di evitare delle chiamate a funzione che occuperebbero risorse. Spesso vengono usati nelle funzioni con il body ridotto.

Listing 2.48: Funzioni Inline

```
inline fun funzione()
```

```
{
    //corpo funzione
}
```

Utilizzo del Tipo Reified nelle Funzioni Inline

A volte nasce la necessità di accedere ai tipi passati come parametri:

Listing 2.49: Esempio tipo di un nodo

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T?
{
    var p = parent
    while(p!=null && !clazz.isIstance(p))
    {
        p = p.parent
    }
    return p as T?
}
.
.
.
treeNode!!.findParentOfType(MioTreeNode::class.java)
```

In questo esempio controlliamo se un nodo è di un certo tipo.

Listing 2.50: Tipo di un nodo con le Funzioni Inline

```
inline fun <reified T> TreeNode.findParentOfType(): T?
{
    var p = parent
    while(p!=null && p !is T)
    {
        p = p.parent
    }
    return p as T?
}
.
.
.
treeNode!!.findParentOfType<MioTreeNode>()
```

Utilizzando le reified, possiamo usare lo stesso concetto in modo meno verboso e più pulito a livello di chiamata.
Nota: I reified possono essere usati esclusivamente sulle inline function

2.4.7 Modificatore tailrec

Il modificatore tailrec permette di prevenire problemi relativi allo stack overflow. Se una funzione è segnata con questo modificatore, la sua chiamata viene ottimizzata in modo tale da avere un ciclo, invece della chiamata a funzione stessa. Una funzione ricorsiva è idonea per la tailrec se la chiamata della funzione a se stessa è l'ultima operazione eseguita. Ad esempio:

Listing 2.51: Funzione non idonea per il tailrec

```
fun factorial (n: Int): Long
{
    if (n == 1)
    {
        return n.toLong()
    }
    else
    {
        return n* factorial (n - 1)
    }
}
```

In questo caso non è idoneo per usare il tailrec poichè *n*factorial(n - 1)* non è l'ultima operazione eseguita all'interno della funzione.

Listing 2.52: Funzione idonea per il tailrec

```
fun fibonacci(n: Int, a: Long, b: Long): Long
{
    return if (n == 0) b else fibonacci(n-1, a+b, a)
}
```

Mentre, in questa occasione, la funzione fibonacci è idonea per usare il modificatore *tailrec*. Per questa ragione, essa può essere implementata nel seguente modo:

Listing 2.53: Funzione che usa il modificatore tailrec

```
tailrec fun fibonacci(n: Int, a: BigInteger, b: BigInteger): BigInteger
{
    return if (n == 0) a else fibonacci(n-1, b, a+b)
}

fun main(args: Array<String>)
{
    val n = 100
    val first = BigInteger("0")
    val second = BigInteger("1")

    println(fibonacci(n, first, second))
}
```

2.5 Programmazione Generica

La programmazione generica consiste nell'implementazione di costrutti che possono essere utilizzati con tipi di dati differenti. Anche se Kotlin è considerato un linguaggio molto tipizzato, certe volte si ha la necessità di creare classi o funzioni generiche.

Listing 2.54: Creazione di una classe Generica

```
class NomeClasse <T> (t:T)
{
    //corpo classe
}
```

In questo modo, quando viene creato un oggetto di questa classe, si può passare come parametro T un oggetto di qualsiasi tipo:

Listing 2.55: Passaggio di un Oggetto qualsiasi

```
val m = NomeClasse<Int>(1)
```

Se il tipo può essere dedotto, allora <...> si può omettere.

Variance

Kotlin fornisce la possibilità, in caso di Generics, di poter passare non solo la classe generica stessa ma anche sottoclassi o superclassi. Per fare ciò è necessario indicare se il tipo è prodotto(*out*) o consumato(*in*) dalla classe:

Listing 2.56: Esempio di una classe in e out

```
class Prodotto <out T>
{
    //corpo classe
}

class Consumato <in T>
{
    //corpo classe
}
```

- in: il tipo viene solo consumato dalla classe, quindi non si possono avere funzioni che hanno come tipo di ritorno il tipo generico. In questo caso il tipo passato può essere anche una sottoclasse del tipo generico, ma non il viceversa. Questo perché all'interno delle funzione possono essere chiamati metodi sul tipo T che possono non essere presenti in una sua superclasse.

- out: il tipo viene solo prodotto dalla classe, quindi può essere passato in ingresso solo al costruttore e non ai metodi. In questo caso il valore ritornato da un metodo può essere salvato anche in una superclasse di T.

2.6 Interoperabilità con Java

2.6.1 Utilizzo di Java su Kotlin

Essendo interoperabile con Java, Kotlin può richiamare le librerie Java in modo naturale.

Listing 2.57: Utilizzo della Classe List di Java

```
import java.util.*

fun prova(lista : List<Int>)
{
    val altra_lista = ArrayList<Int>()
    //ciclo di collection di Java
    for(elem in lista)
    {
        altra_lista.add(elem)
    }
}
```

Metodi Getter e Setter

I metodi Getter e Setter che seguono la convenzione di Java, in Kotlin vengono viste come delle proprietà.
Se una classe Java dispone solo del metodo set, non può essere attribuita come proprietà poiché in kotlin non esiste il concetto di set-only properties.

Listing 2.58: Esempio dell'uso di get e set in Kotlin

```
import java.util.Calendar

fun prova()
{
    val calendar = Calendar.getIstance()

    if(calendar.firstDayOfWeek == Calendar.SUNDAY)
        calendar.firstDayOfWeek=Calendar.MONDAY

    if(!calendar.isLenient)
        calendar.isLenient = true
}
```

In Kotlin, *calendar.isLenient=true* corrisponde ad usare il *setLenient(true)* in Java.

Void

Il ritorno di tipo void in un metodo di Java corrisponde al ritorno di un tipo Unit in kotlin:

Listing 2.59: Esempio dell'uso di Unit in Kotlin

```
import java.util.Calendar

fun prova() : Unit
{
    pritnln("Ciao Mondo")
}
```

Arrays

In Kotlin, come in Java, gli array sono invarianti, cioè tutti gli elementi devono essere dello stesso tipo.

Gli array in Java sono usati per evitare il costo delle operazioni di boxing/unboxing.
Kotlin dispone di classi specializzate per ogni tipo di primitiva da gestire: IntArray, DoubleArray, etc... Essi non sono correlati alla classe Array ma il compilatore sfrutta le primitive di Java per ottenere una massima prestazione in fase di compilazione e uso delle array di Kotlin.

Varargs

In Java viene usato *varargs* per dichiarare, ad esempio, un numero variabile di argomenti in un metodo.
In Koltin, per poter passare un array nella funzione dichiarata con la varargs, si usa l'operatore *spread (*)*:

Listing 2.60: Esempio dell'uso di spread in Kotlin

```
class EsempioJavaArray
{
    fun prova(vararg param: Int)
    {
        //corpo funzione
    }
}
```

Metodi di Objects

In Kotlin *Object* viene indicato con *Any*, e su di esso sono implementati *toString()*, *hashCode()* e *equals()*.
Metodi come *wait()* e *notify()* non possono essere usati per *Any*; nel caso in cui si necessita dell'utilizzo, verranno richiamati in modo esplicito:

Listing 2.61: Uso del metodo notify

```
(obj as java.lang.Object).notify()
```

Mentre, per il ritorno dell'oggetto di una classe Java si usa la notazione seguente:

Listing 2.62: Ritorno di un oggetto della classe Java

```
val objClass = obj::class.java
```

Uso delle parola chiave external

Per dichiarare una funzione che implementa un codice nativo(C o C++), si usa la parola chiave *external* prima di dichiarare la funzione.

2.6.2 Utilizzo di Kotlin su Java

Adesso si andrà ad analizzare la compatibilità di Kotlin su Java.
Il codice di Kotlin può essere chiamato facilmente da Java.

Getter e Setter

Per compilare una proprietà di Kotlin, Java dispone di un metodo getter e di un metodo setter:

Listing 2.63: Getter e Setter di Java

```
public class ClasseJava
{
    private String name;

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name=name;
    }
}
```

Funzioni Package-Level

Tutte le funzioni e proprietà dichiarate in un file kt (quindi linguaggio Kotlin), possono essere richiamati nel seguente modo:

Listing 2.64: Dichiarazione di una funzione in Kotlin

```
package NomeClasse

fun funzione()
{
    //corpo funzione
}
```

Listing 2.65: Uso della funzione di Kotlin in Java

```
import NomeClasse.NomeClasseKt;

public class NomeClasseJava
{
    public static void main()
    {
        NomeClasseKt.funzione();
    }
}
```

Nel primo Listing è stata dichiarata una funzione di Kotlin nel package NomeClasse; mentre nel secondo Listing, la funzione viene richiamata in Java importando il package con il nome della classe implementata in Kotlin e successivamente scrivendo nomeClasse.nomeMetodo.

Esposizione di una proprietà in Java

Per esporre una proprietà di Kotlin come campo in Java, viene utilizzata l'annotazione *@JvmField*:

Listing 2.66: Dichiarazione di una classe in Kotlin con un campo statico

```
package NomeClasse

class NomeClasseKt(id:String)
{
    @JvmField val campo = id
}
```

Listing 2.67: Uso del campo statico di Kotlin in Java

```
import NomeClasse.NomeClasseKt;

public class NomeClasseJava
{
    public String getId(NomeClasseKt nckt)
    {
        return nckt.campo;
    }
}
```

@JvmField crea un campo statico.
Usando, invece, la parola chiave *lateinit* prima della dichiarazione di una variabile (o in un oggetto o in un companion object), si otterrà in Java un campo non statico:

Listing 2.68: Dichiarazione di una classe in Kotlin con un campo non statico

```
package NomeClasse

object NomeClasseKt{
    lateinit var prova : Prova
}
```

Listing 2.69: Uso del campo non statico di Kotlin in Java

```
import NomeClasse.NomeClasseKt;

public class NomeClasseJava
{
    public static void main()
    {
        NomeClasseKt.prova = new Prova();
    }
}
```

Nota: Per l'uso di un metodo statico, sia su un companion object che su di un oggetto, in Kotlin verrà dichiarato usando *@JvmStatic*.

KClass

Può capitare di dover chiamare un metodo di Kotlin, il quale contiene un parametro di tipo Kclass; in questo caso, poichè tale parametro non

era stato pensato per essere utilizzato anche in Java, bisogna invocare, manualmente, l'equivalente di *Class<T>.kotlin*:

Listing 2.70: Uso del parametro KClass in Java

```
import NomeClasse.NomeClasseKt;

kotlin .jvm.JvmClassMappingKt.getKotlinClass(NomeClasseKt.class);
```

Overloads delle funzioni e dei costruttori

Se viene implementata una funzione con valori di default passati come parametri, utilizzando la parola chiave *@JvmOverloads*, verrà generata, per ogni parametro di default, l'overload della funzione:

Listing 2.71: Uso dell'annotazione @JvmOverloads

```
import NomeClasse.NomeClasseKt;

class Prova @JvmOverloads constructor(x: Int, y: Double= 0.0)
{
    @JvmOverloads fun funzione(a: String, b: Int=0, c: String="abc")
    {
        //corpo funzione
    }
}
```

Si otterrà come risultato:

Listing 2.72: Overload ottenuti da @JvmOverloads

```
//Overload dei Costruttori :
Prova(int x, double y)
Prova(int x)

//Overload dei Metodi
void funzione(String a, int b, String c) {}
void funzione(String a, int b) {}
void funzione(String a) {}
```

Controllo delle Eccezioni

Kotlin non ha eccezioni controllate; quindi se si volesse catturare, in Java, un'eccezione sollevata da una funzione di Kotlin, bisogna usare l'annotazione *@Throws*:

Listing 2.73: Uso dell'annotazione @Throws

```
@Throws(IOException::class)
fun funzione()
{
    throw IOException()
}
```

Dal punto di vista di Java, la gestione viene eseguita sempre attraverso il blocco *try/catch*.

Traduzione del tipo Nothing in Java

Poiché il tipo *Nothing* in Java non può essere riconosciuto, Kotlin genera un raw-type usando come tipo di argomento *Nothing*. In questo modo può essere tradotto ed interpretato in Java:

Listing 2.74: Raw-Type del tipo Nothing

```
fun funzione(): List<Nothing> = listOf()

//Essa sara' tradotta in List funzione() {...}
```

2.7 Coroutines

Durante l'esecuzione di un programma capita spesso che venga fatta una chiamata a una funzione che impiega molto tempo per essere eseguita (ad esempio una funzione di I/O o una richiesta di rete). Nella maggior parte dei casi, il thread che ha fatto la chiamata viene messo in attesa fino a quando la funzione non termina. Sospendere un thread è un'operazione costosa e complessa, dato che comporta l'impiego di risorse di calcolo al fine di salvare lo stato del thread di esecuzione e caricare quello del thread che deve essere eseguito; questa operazione si chiama context switch e va fatta ogni volta che cambia il thread in esecuzione da parte del processore.
In Kotlin sono state introdotte le coroutine. Al fine di evitare il blocco di un thread è stata introdotta la sospensione di una coroutine che è più economica e controllabile. Le coroutine semplificano la programmazione concorrente, demandando la complessità delle operazioni alla libreria di sistema e mantenendo una logica sequenziale per il programmatore. Quindi grazie alle coroutine è possibile sospendere l'esecuzione senza bloccare il thread che la esegue.
La libreria si occuperà di inserire le parti rilevanti del codice all'interno di callback, subscrive o eventi e schedulare l'esecuzione del programma su thread differenti, lasciando il codice semplice come se fosse eseguito sequenzialmente. Il tutto è implementato tramite tecniche di compilazione e non è necessario il supporto di virtual machine o sistemi operativi. Il concetto dietro le coroutine è quello di dividere il codice in una sorta di macchina a stati che esegue le operazioni e poi viene messa in stato di attesa prima di poter essere nuovamente richiamata.
I vantaggi dell'uso di coroutine sono principalmente due:

- Vengono utilizzate meno risorse, dato che non è necessario fare nessun context switch. La sospensione di una coroutine è un'operazione quasi gratuita dal punto di vista computazionale.

- Le coroutine sono costruite in modo tale che non possono essere interrotte in un punto casuale del codice (come invece avviene nei thread); è il programmatore che decide in quali punti la coroutine viene interrotta per lasciare spazio ad altre. Questo evita che la coroutine venga fermata in punti in cui una sospensione potrebbe creare problemi al flusso di esecuzione.

2.7.1 Funzioni Suspending

Le coroutine possono essere avviate solo da una funzione suspending e quest'ultime sono le uniche che possono sospendere una coroutines. La dichiarazione è la seguente:

Listing 2.75: Funzioni Suspending

```
suspend fun funzione() {...}
```

Una funzione suspending può essere richiamata solo all'interno di un'altra funzione suspending. Questo comporta che la prima funzione è una lambda o una funzione anonima.

Listing 2.76: Prima chiamata di una funzione suspending

```
suspend fun doWork() { ... }

fun <T> async (block: suspend () -> T) {}

fun main (args: Array<String>)
{
    async{ doWork() }
}
```

Quindi per far partire la funzione *doWork()* che è suspending, si crea una funzione *async* non suspending che accetta come paramentro una funzione che lo è.

2.7.2 Interfaccia CoroutineContext

L'interfaccia CoroutineContext rappresenta il contesto di una coroutine. Contiene un set di elementi indicizzati che rappresentano lo stato di una coroutine al momento dell'interruzione.
La libreria standard di Kotlin non contiene nessuna classe che implementa questa interfaccia; sta quindi al programmatore definire un oggetto che implementa l'interfaccia e inizializzarlo.

Listing 2.77: Interfaccia CoroutineContext

```
interface CoroutineContext
{
    operator fun <E: Element> get (key: Key<E>) : E?
    fun <R> fold( initial : R, operation: (R,Element) -> R) : R
    operator fun plus(context: CoroutineContext) : CoroutineContext
    fun minusKey(key: Key<*>) : CoroutineContext
    {
        val key: Key<*>
    }

    interface Key<*:Element>
}
```

- **get**: da accesso type-safe ad un elemento tramite key
- **fold**: per iterare tutti gli elementi del contesto
- **plus**: rimpiazza gli elementi dell'oggetto con gli elementi passati che hanno la stessa chiave
- **minusKey**: torna un contesto che non contiene la key

2.7.3 Interfaccia Continuation

L'interfaccia Continuation permette di salvare il contesto di una coroutine per poterlo riprendere successivamente

Listing 2.78: Interfaccia Continuation

```
interface Continuation<in T>
{
    val context : CoroutineContext
    fun resume(value:T)
    fun resumeWithException(exception: Throwable)
}
```

- **context**: un oggetto di tipo CoroutineContext che contiene il contesto della coroutine.
- **resume**: callback che fornisce il risultato della coroutine
- **resumeWithException**: callback che riporta un fallimento ritornando un'eccezione

2.7.4 Avviare e fermare una coroutine

La gestione del flusso di esecuzione delle coroutine viene fatto tramite due funzioni di libreria:

- **startCoroutine**: estende una funzione suspending e la avvia come nuova coroutine.

Listing 2.79: Far partire una coroutine

```
1   fun <T> (suspend () -> T).startCoroutine(continuation: Continuation<T>)
```

- **suspendCoroutine**: sospende la coroutine attuale e ne avvia una nuova. Ovviamente può essere chiamata solo all'interno di una funzione suspending

  Listing 2.80: Sospendere una coroutine

  ```
  1   suspend fun<T> suspendCoroutine(block: (Continuation<T>) -> Unit): T
  ```

  Quando una coroutine viene interrotta, viene catturata una continuation che contiene il suo contesto. Quindi in pratica nella continuation è presente il codice che rimane da eseguire e viene passata alla coroutine che sta per partire. In questo modo la coroutine chiamata può ritornare a quella chiamante utilizzando la funzione *continuation.resume()*.

- **createCoroutine**: è la stessa della *startCoroutine*, ma la coroutine creata non viene fatta partire. Ritorna una continuation che contiene tutto il codice da fare eseguire nella coroutine.

2.7.5 CompletableFuture

Il CompletableFuture è una classe che può tornare utile nel caso serva il valore di ritorno di una coroutine asincrona. Nel caso in cui viene lanciata una coroutine in background, si potrebbe avere bisogno a un certo punto di un valore che viene prodotto da essa.
In questo caso viene affidato il valore di ritorno della coroutine a un CompletableFuture che viene poi richiamato nel momento in cui il valore deve essere consumato. Si possono chiamare su questo oggetto le funzioni *complete* o *completeExceptionally* che mettono in pausa l'esecuzione fino a che il valore non è stato calcolato. Questo vuol dire che dopo aver avviato la coroutine asincrona, l'esecuzione continua fino a che non si ha effettivamente bisogno del suo valore di ritorno; solo a quel punto si mette in pausa l'esecuzione per aspettare che il valore sia pronto.

2.7.6 Sperimentale

Al momento della stesura di questa tesi, le coroutine sono ancora in fase sperimentale. Sono state introdotte con Kotlin 1.1 nella libreria

kotlin.coroutines.experimental e proprio per questo gli stessi creatori di Kotlin consigliano di aggiungere il suffisso experimental ai package che contengono coroutine.
Qui è stato descritto il funzionamento di base delle coroutine, ma sono in sviluppo funzioni di libreria che, sfruttando questi metodi di base, hanno come obiettivo quello di mettere, a disposizione degli utenti, strumenti di più alto livello (come ad esempio *async* o *await*). Queste funzioni sono in sviluppo nella libreria *kotlinx.coroutines.experimental*.

Capitolo 3

Sviluppo dell'applicazione

3.1 Scopo dell'applicazione

Lo sviluppo di questa applicazione ha esclusivamente scopo di studio. L'obiettivo è quello di utilizzare in un progetto verosimile di applicazione Android le nozioni imparate precedentemente. In questo modo possono essere rilevate eventuali criticità altrimenti impossibili da scoprire in un contesto teorico, oltre che verificare la maturità effettiva degli strumenti che il linguaggio mette a disposizione.

3.2 Requisiti

Il progetto consiste nella realizzazione di un'applicazione Android per la gestione e lo svolgimento di percorsi turistici attraverso dei punti di interesse.
Sfruttando la geolocalizzazione dello smartphone e ricevendo da un server, sviluppato dal collega Vito Scornavacche, i luoghi registrati, l'applicazione deve fornire all'utente dei punti di interesse nelle vicinanze, dando l'opportunità di seguire dei percorsi calcolati dall'applicazione.

3.2.1 Applicazione Android

L'applicazione si occuperà di inviare la posizione del device al server, il quale tornerà i luoghi di interesse nelle vicinanze dell'utente finale.
Dopodichè, quest'ultimo potrà effettuare una scelta dei luoghi da raggiungere.
I punti di interesse saranno forniti sia con la visualizzazione di una lista, sia con la visualizzazione di una mappa.
Ogni punto di interesse, fornirà informazioni quali: posizione (tramite

la mappa), immagine del luogo, nome ed una breve descrizione.
L'applicazione darà la possibilità, agli utenti, di organizzare un viaggio, impostando luogo di partenza e luogo di arrivo; attraverso questa modalità, verranno mostrate tutte le posizioni che l'utente potrebbe decidere di visitare durante il suo percorso.
Per svolgere le azioni precedentemente citate saranno implementati meccanismi per la gestione della geolocalizzazione (sfruttando le API di Google), sia per fornire la posizione del device, sia per poter usufruire della visualizzazione dei luoghi sulla mappa.

3.3 Progettazione

Nella fase di progettazione rientrano la definizione del comportamento dell'applicazione e degli strumenti da utilizzare.

3.3.1 Richieste HTTP

Bisogna definire il sistema di comunicazione che verrà utilizzato per trasmettere dati tra applicazione e server.

Il protocollo scelto per le richieste fatte al server è HyperText Transfer Protocol (HTTP); è un protocollo a livello di applicazione di tipo richiesta/risposta per informazioni distribuite, collaborative e sistemi ipermediali.

Un client invia una richiesta al server richiamando un metodo su una determinata URI e con un eventuale body; il server risponde con un codice status e un body opzionale. I metodi di richiesta sono:

- **GET**: Richiesta per ottenere dei dati dal server. Non contiene body
- **POST**: Richiesta per creare un nuovo dato su server
- **PUT**: Richiesta per aggiornare un dato sul server
- **DELETE**: Richiesta per eliminare un dato dal server

La risposta del server contiene lo status code: un codice numerico che fornisce informazioni sull'esito della richiesta.
HTTP è il protocollo più usato per le comunicazioni sul web e costituiva il nucleo base del Worl Wide Web insieme ad HTML e URL.

3.3.2 Gradle

Il Gradle è un software open-source per l'automazione dello sviluppo e la gestione delle dipendenze. A partire dal 2013, è il sistema di costruzione automatizzato scelto da google per Android.
Il gradle automatizza le seguenti funzioni:

- compilazione
- packaging
- test automatizzati
- deployment
- documentazione

In questo contesto non deve essere utilizzato in tutte le sue funzioni, dato che non sono oggetto di tesi; è utile però per poter importare delle librerie che sono necessarie allo sviluppo del progetto e all'utilizzo dei vari strumenti.

3.3.3 JSON

Nella comunicazione tra i client e il server, può essere necessario scambiare dei dati strutturati e che possono essere interpretati come oggetti da entrambi gli host. Per fare ciò una valida soluzione è usare il formato JSON; facile da leggere per le persone e veloce da generare e tradurre per i calcolatori.
JSON è costituito di due strutture principali:

- Una lista di coppie chiave/valore
- Una collezione di valori, comunemente definito *array*

L'utilità di usare questa struttura per lo scambio di dati è principalmente fornita, data la sua diffusione, dalla disponibilità di librerie che traducono oggetti in JSON e viceversa, permettendo di fatto lo scambio di oggetti tra client e server.
La libreria utilizzata nel progetto per la traduzione dei JSON è GSON, sviluppata da Google; si può importare sul Gradle col seguente codice:

Listing 3.1: Aggiungere la dipendenza di GSON

```
1 compile "com.google.code.gson:gson:2.8.1"
```

3.3.4 Applicazione Android

Gli strumenti adottati per lo sviluppo dell'applicazione, sono inerenti a due aspetti essenziali che coinvolgeranno il progetto:

- Comunicazione con il server back-end: verrà utilizzando il protocollo HTTP per mezzo del client REST *retrofit*.
- Utilizzo delle *API di Google* inerenti alla geolocalizzazione: attraverso la configurazione del file *Manifest* di Android.

Retrofit

Per la costruzione di richieste http, si utilizzerà un architettura di tipo REST client, consona ed utilizzata in Java per le applicazioni Android: *Retrofit*. E' una libreria, sviluppata dalla Square Open Source, che sfrutta le transazioni rete. E' veloce e facile da utilizzare dal punto di vista di prelevamento e interpretazione della risposta dei dati (come risposta viene generalmente ricevuto un JSON).
Proprio perchè è la prima scelta come libreria di rete per le richieste HTTP di Java, si è pensato di utilizzarla in questa tesi per studiarne la compatibilità con Kotlin.
La caratteristica principale che serve per costruire una richiesta HTTP con refroit risiede nell'uso delle annotazioni.
Le annotazioni, quindi, sono fondamentali per:

- La creazione dei metodi per eseguire una determinata richiesta (GET, POST, PUT, DELETE)
- La creazione del percorso di una determinata richiesta (*@Path*) e il passaggio delle proprietà da usare (*queryParam*)
- La modellazione dell'*Header* e del *Body*

Altro aspetto importante è che retrofit può essere configurato per l'uso di un certo tipo di *convertitore*, il quale gestisce la serializzazione e la deserializzazione dei dati; ad esempio per la gestione dei formati JSON si hanno tre tipi di convertitori: Gson, Jackson, Moshi.
Per poter usufruire, in Android, delle librerie di retrofit, si ha la necessità di aggiungere nel file *Gradle* del progetto, tutte le dipendenze riguardanti il tipo di convertitore, l'adapter (per le *Call* delle istanze) e le RXJava. Quest'ultime sono utili per la programmazione di flussi *observable* (osservabili), utili per gestire la risposta che arriva da parte

del server: un sottoscrittore attende la risposta; essendo osservabile, può esserne interpretato il suo stato (errore o successo da parte del server nella risposta).

Ecco come importare, attraverso il gradle, le dipendenze di retrofit:

Listing 3.2: Aggiungere le dipendenze di Retrofit

```
implementation 'com.squareup.retrofit2:retrofit:2.3.0'
implementation 'com.squareup.retrofit2:converter-gson:2.3.0'
implementation 'com.squareup.retrofit2:adapter-rxjava2:2.3.0'
implementation 'io.reactivex.rxjava2:rxandroid:2.0.1'
```

Android Manifest

Ogni applicazione Android deve avere un file xml chiamato esattamente*AndroidManifest.xml*.
Il file manifest presenta le informazioni essenziali sull'applicazione Android; informazioni che il sistema deve avere prima di poter eseguire qualsiasi codice dell'applicazione.
Le informazioni importanti che devono sicuramente essere considerate sono:

- package dell'applicazione (generalmente è il namespace stesso)
- Tema e descrizione di ogni activity del progetto (label, nome di ogni schermata etc)
- Permessi da abilitare al dispositivo
- *Google API*

Nel caso della realizzazione dell'applicazione in questione, sarà dichiarata una API-KEY per l'utilizzo e la visualizzazione della mappa. Mentre i permessi che verranno abilitati saranno i seguenti:

- *ACCESS COARSE LOCATION*: l'accesso della posizione del device è calcolato utilizzando le celle, i punti di accesso wifi, etc;offre una risposta rapida
- *ACCESS FINE LOCATION*: determina la posizione utilizzano i satelliti (tecnologia GPS); vengono determinate posizioni precise, ma richiede più tempo per la risposta e provoca un ritardo nella determinazione della posizione

- *INTERNET*: utilizzata per avere l'accesso ad internet e di conseguenza utile per lo scambio di informazioni con il server (richieste HTTP)

Listing 3.3: Aggiunta dei permessi nel manifest

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.INTERNET" />
```

Presentazione

Avviando per la prima volta l'applicazione, verrà richiesto di abilitare i permessi di condivisione della posizione, obbligatori per l'uso della stessa app.

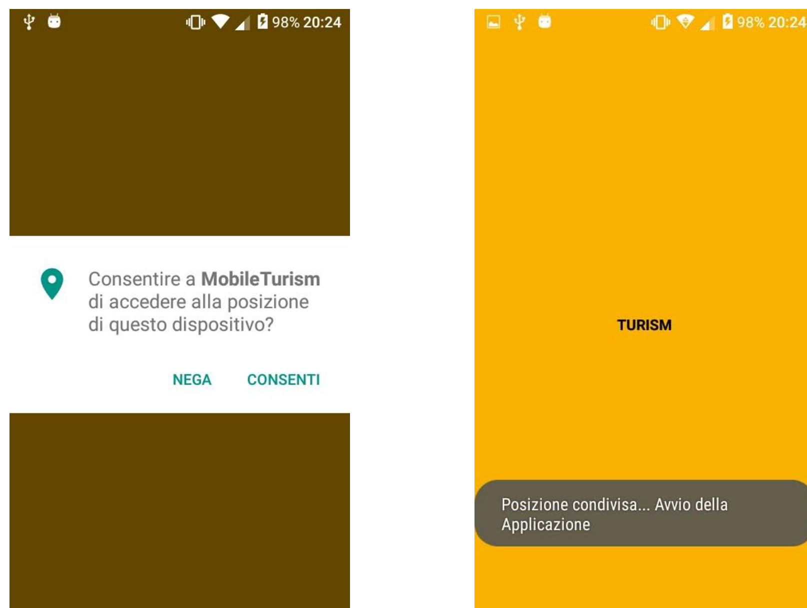

Figura 3.1: Richiesta Accesso Posizione

L'applicazione si presenterà con una schermata sulla quale l'utente avrà la possibilità di effettuare la registrazione oppure, nel caso in cui sia già registrato, il login per accedere alle funzionalità vere e proprie che l'applicativo fornisce.

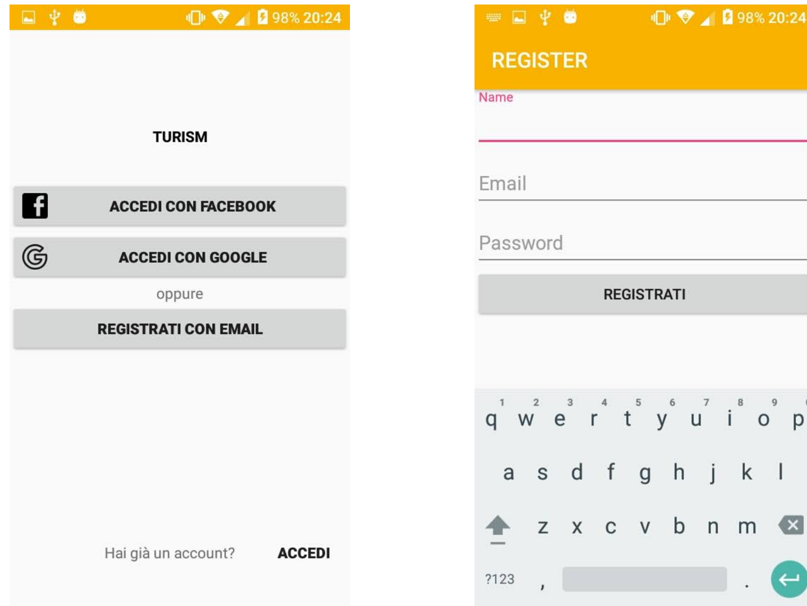

Figura 3.2: Login-Registrazione

Nota: I campi sono tutti obbligatori; la schermata *Login* è analoga a quella di registrazione, senza il nome dello user.
Una volta eseguito il login o la registrazione, le credenziali verranno salvate all'interno del dispositivo, in modo tale che al prossimo riavvio non ci sia la necessità che l'utente debba reinserirle (salvo imprevisti). Successivamente, verrá proposta all'utente una schermata dei luoghi vicini ad esso. Questa activity sarà suddivisa, visibilmente, in tre parti:

- parte superiore: adibita all'uso di un filtro con il quale l'utente, in base alle sue scelte, chiederá solo la visualizzazione delle posizioni che gli interessano, rispetto a quelle tornate dal server.

- parte inferiore: utilizzata per lo switch da una schermata all'altra dell'applicazione.
  Nota: questo layout sarà comune per tutte le activity che lo useranno.

- parte centrale: sicuramente quella principale di questa vista. Essa si occuperà della visualizzazione dei punti di interesse, dando la possibilità al client di scegliere la vista dei luoghi o tramite una

lista (con nome e immagine del luogo in questione) oppure con una mappa, composta dai marker che indicano i luoghi e il punto di localizzazione del dispositivo mobile dell'utente. Nel momento in cui il client selezionerá un punto di interesse (che sia sulla mappa o sulla lista), l'applicazione lo porterá in una view descrittiva del luogo scelto.

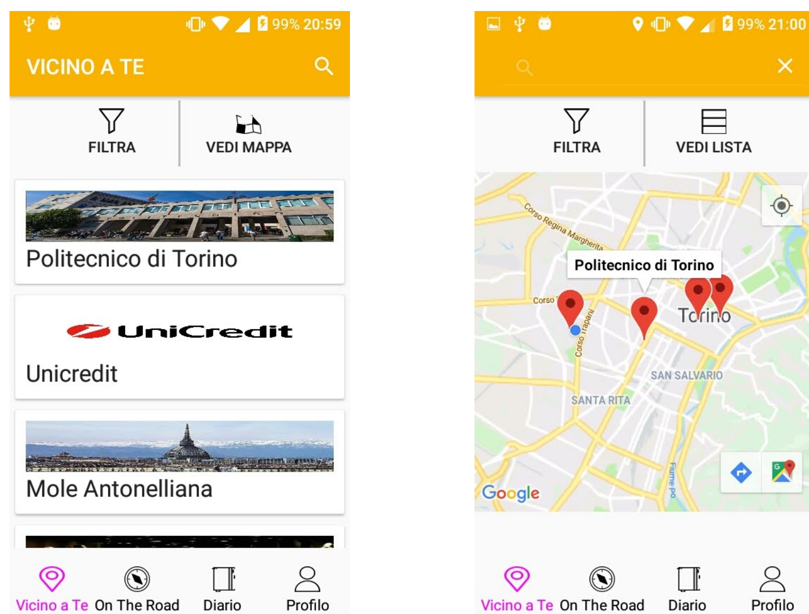

Figura 3.3: Lista/Mappa dei Luoghi Vicini

Nella schermata relativa alla descrizione del luogo selezionato, l'utente finale potrà scegliere definitivamente di essere portato in tale punto. Eseguendo questa opzione, l'applicazione invierà le coordinate, corrispondenti al luogo scelto, a Google Maps, il quale verrà avviato.

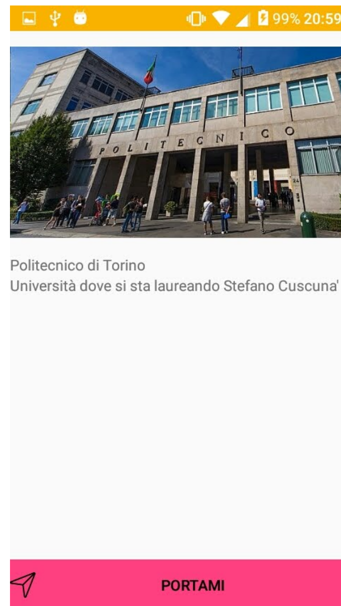

Figura 3.4: Descrizione di un luogo

L'applicativo proporrà, inoltre, una modalità *on the road*, con la quale l'utente potrà creare un suo viaggio. Verrà inserito il luogo di partenza, la destinazione e successivamente tali coordinate verranno inviate al server. La risposta del server sarà una lista di punti di interesse che l'utente potrà trovare nel suo viaggio. Pertanto il client potrà selezionare quali andare a vedere e quali no. Questa scelta verrà tradotta, dall'applicazione, con la visualizzazione della mappa e dei luoghi di interesse scelti dall'utente (oltre che ai punti di partenza e di arrivo), comprensiva di un itinerario in cui l'utente dovrà procedere camminando per raggiungere le varie destinazioni.

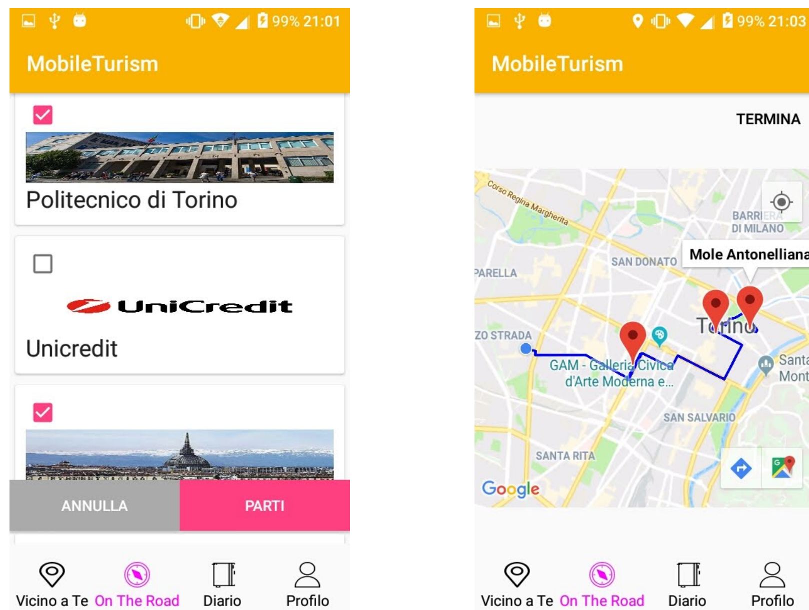

Figura 3.5: Scelta Itinerario Viaggio

Infine sarà sviluppata la schermata *profilo*, nella quale verranno inseriti informazioni inerenti all'utente. Per lo sviluppo di questa tesi è previsto l'inserimento del nome utente, ma si potrebbe pensare, in futuro, di usufruire di tale interfaccia anche per altri scopi; un esempio potrebbe essere quello di espandere la modalità di viaggio, dando la possibilità all'utente di scegliere persino un mezzo di trasporto; un'altra caratteristica interessante che si potrebbe pensare di scegliere è quella riguardante la modalità di come il luogo potrebbe essere descritto, cioè a dire non solo attraverso un testo ma anche con una voce che lo legga.

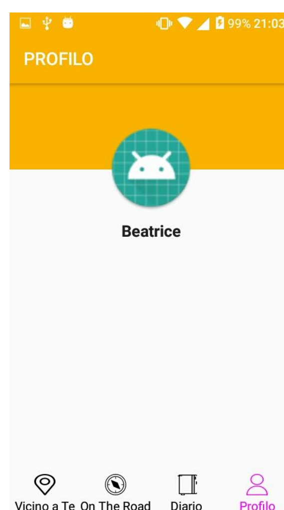

Figura 3.6: Profilo

Problema del Commesso Viaggiatore

Il problema del commesso viaggiatore, spesso noto anche come Travelling Salesman Problem (TSP), è un problema di studio tipico dell'informatica teorica e della teoria della complessità. Il nome nasce dalla sua rappresentazione: dato un insieme di città, connesse da strade, si trovi il percorso minore che il commesso viaggiatore deve percorrere per visitarle tutte e tornare alla città di partenza senza ripercorrere la stessa strada. Si tratta di un problema apparentemente semplice ma in realtà le sue soluzioni sono talmente complesse, in quanto crescono in maniera esponenziale, da rientrare nella classe dei problemi difficili NP-completi (teoria della complessità computazionale). Un eventuale soluzione sarebbe quella di elencare tutti i possibili percorsi e scegliere quella migliore ma la complessità del problema rende tale operazione impraticabile.
Ad oggi sono stati progettati vari algoritmi approssimati, che hanno un alta probabilità di produrre una "buona" soluzione "velocemente"; tuttavia poiché è possibile verificare l'ottimizzazione solo in casi particolari, non si è mai certi di avere la più corretta risoluzione.

3.4 Implementazione

Dopo aver discusso della struttura e delle scelte progettuali dell'applicazione, bisogna procedere verso la realizzazione concreta di essa.

3.4.1 Applicazione Android

Come preannunciato nei paragrafi precedenti, l'applicazione è stata suddivisa in base alla diversificazione delle sue funzionalità: login o registrazione, punti di interesse vicini, on the road e profilo. Dal lato dell'implementazione del codice, si è mantenuta questa suddivisione attraverso l'utilizzo di package differenti; oltre ad aver realizzato i package per le azioni indicate precedentemente, sono stati implementati altri package per la gestione di metodi e dichiarazioni di classi utilizzati in tutto il progetto.
La scelta di una struttura del genere è stata pensata anche per avere una visione più lineare di ogni attività, nonchè una gestione, anche a livello di manutenzione, pulita e rapida per ogni tipo di cambiamento.
Riepilogando, la struttura del progetto è la seguente:

- Package Models: inerente alla dichiarazione delle classi e delle costanti utilizzate nell'intera applicazione
- Package Handler: realizzato per la gestione dei metodi e delle liste e layout grafici comuni nelle varie activity
- Adapter package: per l'amministrazione sulla visualizzazione dei punti di interesse e l'utilizzo dei filtri dell'applicativo
- Package LoginAndRegister: riguardante tutta la parte legata al login e registrazione dell'utente
- Package NearbyPlaces: per le azioni che riguardano i luoghi vicini all'utente finale
- Package OnTheRoad: per l'organizzazione del viaggio del client
- Package Description: relativo alle informazioni fornite di un luogo selezionato dall'utente

Infine, separatamente è stata implementata l'activity che concerne il settaggio del nome dell'utente: ProfileActivity.

3.4.2 Package Models

Questo package è stato imbastito per due ragioni: definizioni delle classi e quindi degli oggetti utilizzati per rappresentare le informazioni attinenti al progetto, e alla implementazione di Retrofit.

UserData

All'interno del file *UserData.kt* si troveranno le costanti e i data class utili al progetto.
In particolar modo si può attenzionare l'uso appropriato dell'ereditarietà tra la classe *Login* e la classe *User*:

Listing 3.4: Ereditarietà Login-User

```
1 //Login
2 open class Login constructor(val mail: String, val pass: String)
3
4 //Estensione di Login
5 class User(mail: String, pass: String, val nome: String) : Login(mail, pass)
```

Un altro aspetto importante di *UserData.kt* è l'implementazione della classe *MyPreferences*. Questa classe sfrutta l'interfaccia *SharedPreferences*, la quale permette l'accesso e la modifica dei dati ritornati da *context.getSharedPreferences(String,Int)*; questo metodo è di tipo thread-safe. Utilizzando questo tipo di meccanismo si possono facilmente salvare delle informazioni (chiave-valore) e recuperarle attraverso l'oggetto *SharedPreferences*. Questa realizzazione è stata studiata per la gestione del nome dell'utente e per il salvataggio delle credenziali.

MobileApi e UnitConverterFactory

Gli altri due file da considerare all'interno del package sono legati all'uso di Retrofit. *MobileApi* è un'iterfaccia sulla quale vengono dichiarati i metodi utilizzati per le richieste HTTP; inoltre, verrà costruito, un oggetto statico, il quale potrà richiamare i metodi per le richieste HTTP:

Listing 3.5: Retrofit Interface

```
1  interface MobileAPI {
2      @POST(value = REGISTER)
3      fun postRegister(@Query("mail") mail:String,
4                       @Query("name") name:String,
5                       @Query("pass") pass:String): Call<Unit>
6
7      @GET(value = POINTONTHEROAD)
8      fun getPointOnTheRoad(@Query("startingLat") sLat:Double,
9                            @Query("startingLon") sLon:Double,
10                           @Query("finalLat") fLat:Double,
11                           @Query("finalLon") fLon:Double,
12                           @Query("mail") mail:String,
13                           @Query("pass") pass:String): Call<ArrayList<PointOfInterest>>
14
15     companion object {
16         fun create(): MobileAPI{
17             val retrofit = Retrofit.Builder()
18                     .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
19                     .addConverterFactory(UnitConverterFactory)
                           .addConverterFactory(GsonConverterFactory.create())
20                     .baseUrl(BASEURL).build()
21             return retrofit.create(MobileAPI::class.java)
22         }
23     }
24 }
```

Nella funzione *create()* si costruisce l'oggetto statico; da notare che esso è stato costruito non solo per ricevere messaggi di tipo JSON, ma anche per ricevere un oggetto di tipo *UnitConverterFactory*.
Questo non è altro che un oggetto custom, realizzato per far fronte ad una limitazione riscontrata su retrofit: se si volesse attendere, dalla parte del client, un messaggio di risposta con body vuoto, l'applicazione sollevava un'eccezione di tipo IllegalArgumentException.

Questo problema è dovuto dal fatto che il client, non potendo gestire risposte con body nullo, non riesce a chiudere la response HTTP; mentre, questo converter, controlla che la response è *Unit* e forza la chiusura:

Listing 3.6: Uniconverter Factory

```
1  object UnitConverterFactory: Converter.Factory() {
2      override fun responseBodyConverter(type: Type?, annotations: Array<out Annotation>?, retrofit: Retrofit?):
           Converter<ResponseBody, *>? {
3          return if(type == Unit::class.java) UnitConverter else null
4      }
5  }
6
7  private object UnitConverter: Converter<ResponseBody, Unit>{
8      override fun convert(value: ResponseBody) {
9          value.close()
10     }
11 }
```

3.4.3 Package LoginAndRegister

Come detto precendentemente, questo package è adibito per effettuare il login dell'utente o la registrazione. In questo package, come nei prossimi che verranno esposti, le attività verranno suddivise anche in base all'esecuzione di azioni in background pilotate con l'uso della classe astratta: *AsyncTask*.
Attraverso questa classe, si continua a mantenere l'uso corretto del thread principale riguardante l'interfaccia grafica dell'applicativo. Una volta che la classe che implementa *AsyncTask* avrà finito la sua elaborazione in background, verranno pubblicati i risultati per l'interfaccia grafica al thread principale. Questo evita che UI thread (user interface) si possa bloccare per attendere la fine di un'operazione lunga.
Nel caso d'uso inerente al progetto, è stata implementata una classe, per ogni package, che si occupa della gestione delle operazioni in background, implementando appunto i metodi della classe astratta presentata pocanzi; nota importante per l'uso di questa classe è che devono essere dichiarati i tipi di oggetti che essa si aspetta in ingresso e che, alla fine delle sue operazioni, tornerà al UI thread. Per questa ragione, sono stati realizzati due data class (nel package precendente):

- MyTask: data class che accetta in ingresso un oggetto di tipo *Any*: in base a che oggetto viene creato e che successivamente deve essere processato in background, ad esempio in questo caso potrebbero essere o l'oggetto *User* o l'oggetto *Login*

- taskResult: che accetta in ingresso essenzialmente un intero, utile per comprendere se l'operazione in background sia andata a buon fine.

Tornando al concetto di *AsyncTask*, i metodi ridefiniti e quindi utilizzati sono i seguenti:

- onPreExecute: richiamato prima che possano partire le operazioni lunghe. In genere è stato ridefinito per abilitare delle progress bar (le quali finiranno la loro visualizzazione nell'ultimo metodo di questa classe).

- doInBackground: utilizzato per eseguire le operazioni più lunghe, in background; esso riceve in ingresso MyTask e restituisce in uscita taskResult; questo risultato passerà al successivo metodo.

- onPostExecute: è colui che riceve la risposta definitiva e che la restituirà al thread principale. In questo contesto, l'uso è il seguente: disabilitazione della progress bar e controllo del risultato tramite il taskResult, in maniera tale da fornire la gestione di casistiche di errore (in questo caso verrà visualizzato un messaggio inerente tale errore) oppure l'ottenimento di un risultato valido (ad esempio con la visualizzazione dei luoghi di interesse per l'utente finale)

Di seguito, si riporta una parte del codice riguardante l'uso dell'*AsyncTask*:

Listing 3.7: Utilizzo di AsyncTask

```
inner class AsyncTaskRunner : AsyncTask<MyTask, String, taskResult>() {
    override fun onPreExecute() {
        super.onPreExecute()

        if (progress != null)
            progress.visibility = View.VISIBLE
    }

    override fun doInBackground(vararg p: MyTask): taskResult {
        return taskResult(exist_account(p[0].any, p[0].requestType, cont), p[0].requestType, null)
    }

    override fun onPostExecute(taskResult: taskResult) {
        super.onPostExecute(taskResult)

        if (progress != null)
            progress.visibility = View.INVISIBLE

        when (taskResult.result) {
            0 -> {
                when (taskResult.type) {
                    LOGIN -> {
                        Toast.makeText(cont.applicationContext, MESS_LOGIN_INTERNAL,
                                Toast.LENGTH_LONG).show()
                        cont.startActivity(Intent(cont.applicationContext, SplashScreen::class.java))
                    }
```

```
                    REGISTER -> Toast.makeText(cont.applicationContext, MESS_REGISTER_INTERNAL,
                        Toast.LENGTH_LONG).show()
                }
            }
            1 -> {
                cont. startActivity (Intent(cont. applicationContext ,
                    MainActivity :: class .java). setFlags(Intent.FLAG_ACTIVITY_NEW_TASK))
            }
            else -> {
                Toast.makeText(cont.applicationContext, SERVER_ERROR, Toast.LENGTH_LONG).show()
            }
        }
    }
}
```

Tra le activity e i file implementati in questo package, l'attenzione sarà focalizzata sull'activity *SplashScreen*.

SplashScreen

Questa è la prima activity che viene richiamata nell'attimo in cui l'utente seleziona l'applicazione.
Inizialmente è stata creata come schermata di avvio, impostando un timer di 5 secondi subito dopo il quale si veniva catapultati nella schermata per effettuare il login o la registrazione. Successivamente, con l'implementazione dello *SharedPreferences*, SplashScreen controlla se vi sono delle credenziali salvate e nel caso in cui sia così, allora inizializzerà un oggetto *Login* con il quale, attraverso l'uso della *AsynckTask*, saranno inviate le credenziale al server tramite una richiesta HTTP:

Listing 3.8: Caso di credenziali salvate

```
if (!MyPreferences(applicationContext).getUsername().isNullOrBlank()
    &&!MyPreferences(applicationContext).getPassword().isNullOrBlank()) {
        LoginRegisterAsyncTask(applicationContext, progressBar = null)
            .AsyncTaskRunner()
            .execute(MyTask(Login(MyPreferences(applicationContext)
                    .getUsername(), MyPreferences(applicationContext)
                    .getPassword()), LOGIN))
}
```

Listing 3.9: Esempio della post per la registrazione

```
val client by lazy {
    MobileAPI.create()
}

var response = client.postRegister(any.mail, any.nome, any.pass).execute()
if (response.code() == 201) {
    return 1
} else if (response.code() == 409) {
    return 0
}
```

Questo codice, invece, rappresenta come avviene una richiesta HTTP con retrofit e come di conseguenza viene gestita la risposta. Importante è notare che la proprietà *client* è di tipo *lazy*, cioè a dire essa

sarà assegnata nel momento in cui verrà utilizzata e non in fase di dichiarazione.
Infine, un'altra caratteristica importante dell'activity riguarda il permesso di utilizzare la localizzazione del dispositivo:

Listing 3.10: Permesso uso localizzazione

```
private fun checkLocationPermission() {
    if (ContextCompat.checkSelfPermission(this,
        Manifest.permission.ACCESS_COARSE_LOCATION)
            !=PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this,
                arrayOf(Manifest.permission.ACCESS_COARSE_LOCATION),
                PERMISSION_REQUEST_LOCATION)
    } else {
            startApp() //permesso accettato
        }
}
```

Con questa condizione, nel caso in cui ancora non è stata accettata da parte dell'utente, verrà richiesto ad esso il permesso dell'uso della localizzazione del dispositivo. Nel caso in cui l'utente non dovesse accettare, allora l'applicazione verrà chiusa (poichè essa è principalmente basata su questo).
Per il controllo di tale funzionalità è stato ridefinito il metodo *onRequestPermissionResult*:

Listing 3.11: Permesso uso localizzazione2

```
override fun onRequestPermissionsResult(requestCode: Int,
    permissions: Array<String>, grantResults: IntArray) {
    when (requestCode) {
        PERMISSION_REQUEST_LOCATION -> {
            if ((grantResults.isNotEmpty()
                        && grantResults[0] == PackageManager.PERMISSION_GRANTED)) {
                printToastMessage("Posizione condivisa... Avvio della Applicazione",
                    this)
                startApp()
            } else {
                AlertDialog.Builder(this)
                        .setTitle("Chiusura della Applicazione")
                        .setMessage("Permesso negato! Devi condividere la tua posizione!!")
                        .setCancelable(false)
                        .setPositiveButton("OK",
                            DialogInterface.OnClickListener { dialog, i ->
                                finish()
                                System.exit(EXIT_CODE)
                            }).create().show()
            }
        }
    }
}
```

3.4.4 Package NearbyPlaces

Questa parte dell'applicativo si occupa principalmente di:

- data la localizzazione del device, viene inviata la richiesta per ottenere i punti di interesse vicini all'utente.

- utilizzo di un filtro per visualizzare, in tempo reale, i luoghi che più gli interessano (tra quelli vicini)
- swap tra visualizzazione della mappa e quella della lista dei luoghi di interesse

Localizzazione del dispositivo e Filtro dei luoghi vicini

La localizzazione del dispositivo viene sfruttata ai fini di ottenere i luoghi di interesse vicini all'utente finale.
Per il conseguimento di tale localizzazione, all'interno del package *Handler* è stata implementato un metodo che riceve in ingresso una funzione; all'interno di questo metodo viene sfruttato il nuovo meccanismo dei servizi di Google Play, utilizzato per ottenere l'ultima posizione recente:

Listing 3.12: Ritorno ultima posizione del device

```
fun positionListener (context: Context, function :() ->Unit){
    var lastLocation = LocationServices. getFusedLocationProviderClient (context)

    if (ContextCompat.checkSelfPermission(context,
                Manifest. permission . ACCESS_COARSE_LOCATION) ==
                    PackageManager.PERMISSION_GRANTED) {
        lastLocation . lastLocation .addOnSuccessListener{ location ->
            if ( location != null) {
                devicePosition = location
                function ()
            } else {
                Toast.makeText(context, ENABLE_POSITION, Toast.LENGTH_LONG).show()
                context. startActivity ( Intent ( Settings . ACTION_LOCATION_SOURCE_SETTINGS))
            }
        }
    }
}
```

La condizione viene gestita nel seguente modo: nel caso in cui la localizzazione del dispositivo è disabilitata, allora l'utente avrà la possibilità di abilitarla nei settaggi del dispositivo (i quali compariranno su di una finestra). Nel momento in cui essi saranno abilitati nel cellulare, allora verrà aggiunto un listener che si occuperà di prelevare la posizione del telefono e utilizzarla per i suoi scopi. Una volta fatto questo, verrà richiamata la funzione *function()*, passata come parametro, che si occupa di eseguire la richiesta HTTP per ottenere i luoghi di interesse, sfruttando la posizione del dispositivo ottenuta:

Listing 3.13: Funzione che richiama l'AsyncTask per la richiesta dei luoghi

```
private fun getNearPoints(): Unit {
    MainActivityAsyncTask(POSITION, this, main_progress,
            findViewById<RecyclerView>(R.id.recyclerView))
            .AsyncTaskRunner()
            .execute(MyTask(Coordinates(getDevicePosition(). latitude ,
                    getDevicePosition (). longitude ), POSITION))
```

```
7 }
```

Questa è la funzione passata come parametro al metodo *positionListener*.

Listing 3.14: Funzione GET e Recycler per la visualizzazione dei luoghi

```
fun requestPointOfInterest(any: Any, cont: Context): Int {
    if (any is Coordinates) {
        val myPreferences = MyPreferences(cont)
        val username = myPreferences.getUsername()
        val passwd = myPreferences.getPassword()
        var response = client.getPointOfInterest(any.lat,any.lon, username, passwd).execute()
        .
        .
        .
}

fun viewPlaces(recyclerView: RecyclerView, type: String, cont: Context) {
    recyclerView.layoutManager = LinearLayoutManager(cont, LinearLayout.VERTICAL, false)
    adapter = CustomAdapter(getList(), cont, type, null)

    recyclerView.adapter = adapter
}
```

La prima funzione viene impegata per reperire la *response* contenente la lista dei luoghi; questa lista verrà successivamente salvata su un *ArrayList <PointOfInterest>* (oggetti contenenti il nome, la descrizione, le coordinate dei punti di interesse e l'immagine del luogo). La seconda funzione viene utilizzata per visualizzare graficamente i sopracitati luoghi, attraverso la realizzazione di una *recycler view*. Per la creazione di questo tipo di container è stata definita una classe a parte, sulla quale viene implementato il metodo *RecyclerView.Adapter<...>*. Questo tipo di *Adapter* fornisce un'associazione tra il set di dati (in questo caso l'Array dei punti di interesse) e la visualizzazione nella lista delle card della *recycler view*. L'Adapter è stato definito all'interno del package *Adapter*:

Listing 3.15: Custom Adapter per la recycler view e il Filter

```
class CustomAdapter(val placeList: ArrayList<PointOfInterest>, val mcontext: Context, type: String, var
    idLinearLayout: LinearLayout?) : RecyclerView.Adapter<CustomAdapter.ViewHolder>(),
        Filterable {

        override fun getFilter(): Filter {
        if (mFilter == null) {
            filteredList.clear()
            filteredList.addAll(mList)
           mFilter = CustomAdapter.MyFilter(this, filteredList)
        }
        return mFilter!!
    }

    override fun getItemCount(): Int {
        return placeList.size
    }

    override fun onCreateViewHolder(parent: ViewGroup?, viewType: Int): ViewHolder {
        var v: View? = null
        if (typeOfCard.equals(POSITION)) {
```

```
20             v = LayoutInflater.from(parent?.context).inflate(R.layout.list_layout, parent, false)
21         } else {
22             v = LayoutInflater.from(parent?.context).inflate(R.layout.list_layout_road, parent, false)
23         }
24 
25         return ViewHolder(v!!, typeOfCard)
26     }
27 
28     override fun onBindViewHolder(holder: ViewHolder?, position: Int) {
29         val pointOfInterest: PointOfInterest = placeList[position]
30 
31         holder?.textViewName?.text = pointOfInterest.name
32         if (holder!!.cardViewButton != null) {
33             holder!!.cardViewButton!!.setOnClickListener {
34                 mcontext.startActivity(Intent(mcontext.applicationContext,
                       DescriptionActivity::class.java).putExtra(POSITION, pointOfInterest.name))
35             }
36         } else {
37             holder!!.checkBox!!.setOnClickListener {
38                 if (it.id_checkBoxRoad.isChecked) {
39                     getListOnTheRoad().add(pointOfInterest)
40                 } else {
41                     getListOnTheRoad().remove(pointOfInterest)
42                 }
43                 if (getListOnTheRoad().isEmpty()) {
44                     idLinearLayout!!.visibility = View.INVISIBLE
45                 } else {
46                     idLinearLayout!!.visibility = View.VISIBLE
47                 }
48             }
49         }
50         .
51         .
52         .
53     }
54 
55     class ViewHolder(itemView: View, type: String) : RecyclerView.ViewHolder(itemView) {
56         var cardViewButton: Button? = null
57         var textViewName: TextView? = null
58         var imageView: ImageView? = null
59         var checkBox: CheckBox? = null
60 
61         init {
62             if (type.equals(POSITION)) {
63                 imageView = itemView.findViewById<ImageView>(R.id.imageViewList)
64                 textViewName = itemView.findViewById<TextView>(R.id.textViewName)
65                 cardViewButton = itemView.findViewById<Button>(R.id.cardViewButton)
66             } else {
67                 imageView = itemView.findViewById<ImageView>(R.id.imageViewListRoad)
68                 textViewName = itemView.findViewById<TextView>(R.id.textViewNameRoad)
69                 checkBox = itemView.findViewById<CheckBox>(R.id.id_checkBoxRoad)
70             }
71         }
72     }
73 
74     class MyFilter(val customAdapter: CustomAdapter, val filteredList: ArrayList<PointOfInterest>) : Filter() {
75         var originalList = ArrayList<PointOfInterest>()
76         var filterList = ArrayList<PointOfInterest>()
77         var myAdapter: CustomAdapter? = null
78         .
79         .
80         .
81         override fun performFiltering(p0: CharSequence?): FilterResults {
82             filterList.clear()
83 
84             val reults = FilterResults()
85             if (p0!!.length == 0) {
86                 filterList.addAll(originalList)
87             } else {
88                 val filterPattern = p0.toString().toLowerCase().trim()
89                 for (point in originalList) {
90                     if (point.name.toLowerCase().contains(filterPattern)) {
91                         filterList.add(point)
```

```
                    }
                }
            }

            reults.values = filterList
            reults.count = filterList.size
            return reults
        }

        override fun publishResults(p0: CharSequence?, p1: FilterResults?) {
            myAdapter!!.mList.clear()
            myAdapter!!.mList.addAll(p1!!.values as ArrayList<PointOfInterest>)
            myAdapter!!.notifyDataSetChanged()
        }
    }
}
```

In questo file è stato definito tutto il meccanismo per la gestione delle *Recycler View* (sia dei luoghi vicini sia per l'on the road) e per il filtro della schermata dei luoghi vicini.

Swap Mappa-Lista dei luoghi di interesse

Per effettuare lo switch, nello stesso layout, tra una schermata ed un'altra, uno dei meccanismi più usati nel mondo Android è il *ViewFlipper*. Essa non è altro che una classe che estende la ViewAnimator. Il suo comportamento è molto significativo: viene realizzata una ViewAnimator, che rappresenterà dei child; ad ogni child viene associata una schermata. Lo switch per passare da un child ad un altro può essere attribuito o tramite dei button oppure anche in sequenza temporale (impostata a livello di codice):

Listing 3.16: Swap Mappa-Lista usando ViewFlipper

```
/** Listener per la gestione dei pulsanti della bottom_layout e della lista e mappa
**/
private fun buttonsListener() {
    id_maps.setOnClickListener {
        id_viewFlipper.displayedChild = 1

        val mapFragment = supportFragmentManager
                .findFragmentById(R.id.map) as SupportMapFragment
        mapFragment.getMapAsync(this)
    }

    id_list.setOnClickListener { id_viewFlipper.displayedChild = 0 }

    getRadioGroupListener(this, id_vicino_a_te, id_on_the_road, id_diario, id_profilo, VICINOATE)
}
```

Qui, si nota l'associazione tra il pulsante ***id_maps*** e il pulsante ***id_list***.
Per la gestione della schermata della mappa, nella classe principale del package (*MainActivity*) è stata implementata l'interfaccia *OnMapReadyCallback*, dopodichè è stato ridefinito il metodo per la gestione della mappa stessa: *onMapReady*.

Listing 3.17: Metodo per la visualizzazione della mappa

```
/** Override del metodo per la visualizzazione della mappa
**/
override fun onMapReady(googleMap: GoogleMap) {
    if (ContextCompat.checkSelfPermission(this,
        Manifest.permission.ACCESS_COARSE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
            googleMap.isMyLocationEnabled = true
            for (point in getList().iterator()) {
                var position = LatLng(point.lat, point.lon)
                googleMap.addMarker(MarkerOptions().position(position).title(point.name))
                googleMap.moveCamera(CameraUpdateFactory.newLatLng(position))
            }
            googleMap.setOnInfoWindowClickListener {
                startActivity(Intent(this, DescriptionActivity::class.java).putExtra(POSITION, it.title))
            }
    }
}
```

In questo metodo si nota l'inserimento dei marker, prendendo le posizioni dall'array dei punti di interesse. Oltre a questo è stato gestito il meccanismo per passare nell'Activity *Description*, con la quale l'utente potrà vedere nome, descrizione e immagine del luogo e potrà decidere di andare lì, chiamando esternamente google maps e passando ad esso le coordinate del luogo:

Listing 3.18: Metodo nel package Description per richiamare Google Maps

```
fun setButton(pointOfInterest: PointOfInterest, cont: Context, id_button_description: Button) {
    val latitude = pointOfInterest.lat
    val longitude = pointOfInterest.lon
    val label = pointOfInterest.name + " : " + pointOfInterest.description
    val uriBegin = "geo:$latitude,$longitude"
    val query = latitude.toString() + "," + longitude + "(" + label + ")"
    val encodedQuery = Uri.encode(query)
    val uriString = "$uriBegin?q=$encodedQuery&z=16"
    val uri = Uri.parse(uriString)
    val intent = Intent(Intent.ACTION_VIEW, uri)
    id_button_description!!.setOnClickListener {
        cont.startActivity(intent)
    }
}
```

3.4.5 Package OnTheRoad

La sezione *On The Road* dell'applicazione riguarda la possibilità che viene data all'utente di impostare un viaggio da compiere. Il concetto implementativo rimane presocchè simile a quello del package *Nearby*: si parla sempre di ottenere una lista di luoghi, ma questa volta inerente a due posizioni collegate tra di loro e non ad un punto. Per questo una parte iniziale dove si nota una variazione rispetto al package precedente è che l'utente deve inserire il luogo di partenza e quello di destinazione; per ottenere tale risultato sono stati implementati due classi *PlaceAutocompleteFragment*, le quali estendono la classe *Fragment*. Questa classe permette il completamento automati-

co della selezione dei luoghi. Con questo strumento si otterranno le coordinate necesserie per eseguire la richiesta HTTP e per ottenere di conseguenza i luoghi che l'utente può trovare mentre affronta il suo viaggio.

Listing 3.19: Uso di PlaceAutocompleteFragment

```
/** Metodo per impostare/cancellare il punto di partenza e di
 *  arrivo su on the road
 **/
private fun searchCoordinates(id: Int) {
    val autocompleteFragment = fragmentManager.findFragmentById(id) as PlaceAutocompleteFragment
    val autoCompleteFragmentView = autocompleteFragment.getView() as ViewGroup
    val clearButtonId = com.google.android.gms.R.id.place_autocomplete_clear_button
    val mClearAutoCompleteButton = autoCompleteFragmentView.findViewById(clearButtonId) as View

    mClearAutoCompleteButton.setOnClickListener {
        if (id == R.id.id_partenza) {
            partenza = null

        } else {
            arrivo = null
        }
        autocompleteFragment.setText(EMPTY_STRING)
    }

    autocompleteFragment.setOnPlaceSelectedListener(object : PlaceSelectionListener {
        override fun onPlaceSelected(place: Place) {
            if (id == R.id.id_partenza) {
                partenza = PointOfInterest( place.latLng.latitude, place.latLng.longitude,
                        place.name.toString(), place.address.toString(), EMPTY_STRING)


            } else {
                arrivo = PointOfInterest( place.latLng.latitude, place.latLng.longitude,
                        place.name.toString(), place.address.toString(), EMPTY_STRING)
            }
        }

        override fun onError(status: Status) { Log.i("ERRORE", "Errore: $status")}
    })
}
```

Viene eseguita anche la cancellazione nel momento in cui si voglia cambiare uno dei due luoghi. Oltre a questo viene anche eseguito il controllo se sono stati impostati tutti e due i luoghi e dopodichè può essere inviato al server. La parte della visualizzazione della lista è sempre gestita nel package *Adapter*, illustrata precedentemente nel Listing 3.15.
Altra differenza rispetto al package precedente è relativa alla visualizzazione della mappa: dopo che vengono restituiti i luoghi, l'utente avrà la possibilità di selezionarli (attraverso una checkbox inserita in ogni card view della recycler view). Una volta selezionati tutti i luoghi da visitare durante il tragitto, compariranno su di una mappa. In questo caso, la gestione della mappa è differente da prima poichè il luoghi vengono ordinati in base alle coordinate di partenza dell'utente.

Listing 3.20: Ordinamento della lista On the Road

```
//Ordina la lista on the road partendo come punto di riferimento il luogo di partenza
Collections.sort(getListOnTheRoad(), SortPlaces(LatLng(location.latitude, location.longitude)))
.
.
.
/** Classe implementata per l'ordinamento dei punti di interesse su on the road
**/
class SortPlaces(internal var currentLoc: LatLng) : Comparator<PointOfInterest> {
    override fun compare(place1: PointOfInterest, place2: PointOfInterest): Int {
        val lat1 = place1.lat
        val lon1 = place1.lon
        val lat2 = place2.lat
        val lon2 = place2.lon

        val distanceToPlace1 = distance(currentLoc.latitude, currentLoc.longitude, lat1, lon1)
        val distanceToPlace2 = distance(currentLoc.latitude, currentLoc.longitude, lat2, lon2)
        return (distanceToPlace1 - distanceToPlace2).toInt()
    }

    fun distance(fromLat: Double, fromLon: Double, toLat: Double, toLon: Double): Double {
        val radius = 6378137.0   // approximate Earth radius, *in meters*
        val deltaLat = toLat - fromLat
        val deltaLon = toLon - fromLon
        val angle = (2 * Math.asin(Math.sqrt(
                Math.pow(Math.sin(deltaLat / 2), 2.0) + Math.cos(fromLat) * Math.cos(toLat) *
                        Math.pow(Math.sin(deltaLon / 2), 2.0))))
        return radius * angle
    }
}
```

Per eseguire l'ordinamento della lista si è creata una classe che implementa *Comparator*; utilizzando questa interfaccia è stato possibile ridefinire il metodo *compare* per ordinare le varie posizioni. Come metodo adottato per l'ordinamento delle posizioni è stato scelto il metodo *Great-circle distance*: è la distanza più breve tra due punti sulla superficie di una sfera, misurata lungo la superficie della sfera. Poichè il progetto verte ai fini di studio del linguaggio, non vi è stata una focalizzazione per implementare un algoritmo migliore rispetto a questo; inoltre bisogna sempre prendere in considerazione il problema del commesso viaggiatore (esposto precedentemente).
Infine, all'interno della mappa, prendendo in cosiderazione un punto di interesse e il successivo punto, è stato realizzato il tracciamento del percorso da eseguire a piedi.

Listing 3.21: Creazione percorso sulla mappa

```
var points: ArrayList<LatLng>? = null
var polyLineOptions: PolylineOptions? = null

// traversing through routes
for (i in 0..result!!.size - 1) {
    points = ArrayList<LatLng>()
    polyLineOptions = PolylineOptions()
    var path: List<HashMap<String, String>> = result.get(i)

    for (j in 0..path.size - 1) {
        var point: HashMap<String, String> = path.get(j)

        var lat: Double = point.get("lat")!!.toDouble()
        var lng: Double = point.get("lng")!!.toDouble()
        var position = LatLng(lat, lng)
```

```
16
17            points.add(position)
18        }
19
20        polyLineOptions.addAll(points)
21        polyLineOptions.width(4f)
22        polyLineOptions.color(Color.BLUE)
23    }
24
25    mMap.addPolyline(polyLineOptions)
26 }
```

In questa parte di codice viene mostrato come è stata aggiunta una linea che congiunge due posizioni. Si evince dal codice che per realizzare il percorso si è utilizzato l'oggetto di tipo *PolylineOptions*; su di esso vengono aggiunti i punti e poi verrà aggiunto lui stesso nella lista dei Polyline che dispone l'oggetto *mMap*(che rapprensenta la mappa). La parte inerente alla costruzione della linea si trova all'interno del file *PathJSONParser.kt* dello stesso package.

Capitolo 4

Valutazione delle criticità

4.1 Esito dello sviluppo in Kotlin

L'interoperabilità di Java, dovrebbe essere utile per evitare di avere incompatibilità future in Kotlin.
Le *Funzioni estensione* e le *High Order Function*, realizzate in Kotlin, rendono questo linguaggio più scalabile ed estensibile.
Attraverso i *data class*, alle *infix function* e a tutti gli strumenti che il linguaggio mette a disposizione rispetto agli altri (e sopratutto in paragone con Java), il codice è reso molto più conciso.
Grazie a tutte queste caratteristiche, potrebbe essere il futuro per Android, ma come tutti i linguaggi, presenta dei pro e dei contro, messi in luce anche attraverso la realizzazione di questa applicazione.

4.2 Kotlin: Vantaggi e Svantaggi

4.2.1 Vantaggi

Come caratteristiche positive e già discusse nella storia dell'arte del linguaggio abbiamo sicuramente la gestione dei puntatori a null, l'accesso alle proprietà e uso delle eccezioni non controllate.
Di seguito verranno descritte ulteriori caratteristiche, utilizzate anche nel progetto.

Delegates.notNull, lazy e lateinit

L'utilizzo e il controllo dei puntatori a null in Kotlin è molto utile; però per quanto riguarda il ciclo di vita di un'Activity in Andorid, bisogna ugualmente inizializzare una proprietà, ed inoltre bisogna considere la dichiarazione delle proprietà a seconda che siano dichiarate come *val*

oppure come *var*.
Nel metodo *onCreate*, infatti non possiamo dichiarare una proprietà *val*, visto che poi non può essere mutabile, ma deve essere dichiarata sia mutabile (*var*) che di tipo *nullable*:

Listing 4.1: Tipi Nullable

```
var nome: String?=null
```

Utilizzando una soluzione del genere, ogni qualvolta si debba accedere alla variabile *nome* si dovranno mettere *!!* per la gestione dei null in Kotlin. Nel progetto è stato utilizzato un approccio del genere, ma altri tipi di approcci possono essere ad esempio:

Listing 4.2: lateinit e Delegates.notNull

```
 lateinit var nome: String
var num: Int by Delegates.notNull<Int>()
```

Così si può definire la proprietà anche come tipo non nullo e non inizializzato. Nota: in entrambi i casi se si prova ad accedere alla proprietà prima che essa sia stata inizializzata, verrà scatenata un'eccezione. Per quanto riguarda la parola chiave *lazy* è stata utilizzata e spiegata su Retrofit.

Estensione di funzioni

Il linguaggio dispone di metodi per l'estensione di funzione ed utilizzo di collezione di iteratori migliorate rispetto che all'utilizzo di Java: any, joinToString e associate sono funzioni ad esempio molto utili per evitare cicli di codifica.

Linguaggio procedurale e funzionale

Come tutti i linguaggi di programmazione moderna, Kotlin cerca di offrire le migliori caratteristiche sia di una programmazione funzionale che di una programmazione procedurale.

Compattezza del codice

Mettendo a confronto un metodo scritto in Java che porta un risultato analogo di un metodo scritto in Kotlin, salterà ad occhio come quest'ultimo risulta essere intuitivo e facile da leggere (anche nelle situazioni più critiche è molto più chiaro).
La visibilità del codice quindi sarà più compatta e succinta; in questo

modo. Ottenendo un codice più snello e pulito si avranno possibilità minori sulla creazione di eventuali bug.

4.2.2 Svantaggi

Android mette a disposizione molte librerie proficue. Però al passaggio verso Kotlin, non sempre tutto va per il verso giusto.

Annotazioni

Nel caso dell'uso delle annotazioni, se si volessero utilizzare librerie come *Icepick* oppure *EventBus*, Kotlin non riesce a rilevare annotazioni come *@State* e *@Subscribe*. Ovviamente, questo non significa che tutte le librerie che usano le annotazioni non funzionino in Kotlin (vedi Retrofit), però questo esempio è un messaggio chiaro che esprime che ancora non si ha una piena compatibilità.

Conversione da Java a Kotlin

Il plugin di Kotlin in Android Studio, rende disponibile un meccanismo per la conversione da codice Java a codice Kotlin, ma questa conversione non è propriamente precisa:

Listing 4.3: Esempio di codice Java da convertire in Kotlin

```
public class NomeClasse {
    String str = "";

    public NomeClasse(String str) {
        this.str = str;
    }
}
```

Eseguendo la conversione tramite il plugin si otterrà:

Listing 4.4: Conversione in Kotlin

```
class NomeClasse(str: String) {
    internal var str = ""

    init {
        this.str = str
    }
}
```

Invece, se si pensava di scriverlo in Kotlin, senza l'uso della conversione, molto probabilmente si sarebbe adottato un approccio del genere:

Listing 4.5: Conversione in Kotlin Ottimale

```
class NomeClasse(val str: String) {}
```

Il problema dell'utilizzo di questo plugin non risiede solo da un punto di vista di conversione ottimale o no, ma questo tipo di conversione potrebbe provocare la generazione di un codice Kotlin non compilabile.
Nota: Essendo ad ogni modo compatibile con Java, si potrebbe aspettare in futuro un plugin che permetta una conversione da Kotlin a Java (per il momento questa conversione automatica non è disponibile).

Scorciatoie da tastiera

Molto spesso, programmando in Java, capita di concludere la scrittura di una funzione o di una parte del codice utilizzando il completamento dell'istruzione, attraverso la sequenza *Alt-Invio*. Sfortunatamente, per il momento in Kotlin queste *shortcuts* non sono utilizzabili.

Refactoring Modulare

Non è detto che funzioni sempre un refactoring per portare un codice ad essere di tipo modulare.

Sintassi delle proprietà d'accesso

Kotlin permette di richiamare i metodi getter e setter come se fossero delle vere e proprie proprietà:

Listing 4.6: Getter in Kotlin

```
1 activity .getContext()
2 .
3 .
4 .
5 activity .context
```

E' un'ottima caratteristiche, infatti alcune volte anche il plugin di Kotlin da come suggerimento di usare il secondo modo piuttosto che il primo. Alcune volte questo tipo di metodo non è chiaro perchè la proprietà potrebbe avere un nominativo differente e quindi poco intuitivo per chi programma.

Oggetti e metodi statici

Rispetto a Java, Kotlin dichiara i metodi e le proprietà statiche in maniera più prolissa.

Metodo del conteggio

Se da una parte è un bene scrivere poco codice con Kotlin, in Android questo può essere un'arma a doppio taglio. La riduzione di codice si trasforma in un aumento del conteggio dei metodi del codice compilato.
Questo problema è più sensibile in Android perchè ogni volta che si vuole definire un campo, essa si tramuterà nella dichiarazione di una proprietà (con l'utilizzo di *val* e *var*). Il vantaggio da una parte sta che per ogni proprietà si ha un getter e setter (getter nel caso in cui sia stato dichiarato con *val*); il problema è che per ogni dichiarazione pubblica delle proprietà ci sarà un aumento del conteggio dei metodi. Questo vuol dire che se in Java si era abituati a dichiarare campi pubblici, in Kotlin bisogna pensarci prima di dichiararli.

Capitolo 5

Conclusioni e Sviluppi Futuri

Come tutti i linguaggi realizzati, si è potuto notare che Kotlin mostra alcuni svantaggi (elencati nel capitolo precedente); sembra però che la maggior parte di essi non sembrino minimamente un ostacolo per un ulteriore sviluppo del linguaggio.
Poichè è stato realizzato da JetBrains e poichè questa società sviluppò Android Studio, in generale è veramente difficile poter affermare problemi significativi che potrebbero escludere un ipotetico sviluppo di un programma in tale linguaggio: si può quindi affermare pienamente un eccellente supporto su Android Studio.
Risulta essere un linguaggio molto meno prolisso rispetto a Java.
La maggior parte delle librerie standard utilizzata nei linguaggi è già nativa in Kotlin.
Anche se non è molto accurata in certi aspetti, la conversione automatica da codice Java a codice Kotlin è veramente utile.
Dallo studio effettuato attraverso la realizzazione di questa applicazione si può evincere che il linguaggio Kotlin non è ancora così maturo per certi punti di vista. Una delle ragioni che porta all'affermazione precedente è che non ancora tutte le librerie utili hanno una compatibilità totale con il linguaggio (ad esempio Retrofit non supporta response con body nullo); oltretutto essendo ancora un linguaggio pressocchè nuovo, non si trovano molte guide che trattano casistiche complesse da implementare.
Un altro aspetto da considerare è che uno sviluppatore può programmare in Kotlin senza essere a conoscenza della programmazione in Java; tuttavia possedendo una conoscenza del genere (e sapendo quindi quali sono i suoi limiti) si possono evidenziare maggiormente le qualità di Kotlin.
Un ulteriore considerazione da fare riguarda il fatto che per la crea-

zione di un determinato algoritmo, lo sviluppatore potrebbe pensare o anche cercare delle guide in Java per poi tradurle in Kotlin.
Attraverso lo studio di questo linguaggio si può evincere che esso sia ancora in via di sviluppo; per quello che è stato studiato attraverso lo sviluppo di questa applicazione, sembra che Kotlin prometti abbastanza bene. Il consiglio che si potrebbe dare sta nel controllare se vi sono delle criticità da migliorare o meno. Nel caso d'uso specifico un suggerimento potrebbe essere quello di ampliare un applicazione del genere, magari sviluppando un meccanismo che tenga traccia delle foto e dei file multimediali del viaggio che l'utente sceglie di fare (Sezione *Diario* del programma non ancora sviluppata); oppure facendo in modo che l'utente possa sentire la descrizione del luogo e non più leggerla. Infine si potrebbe pensare anche ad una scelta di raggiungimento della posizione anche con i mezzi e non solo a piedi.
In conclusione, sfruttando tutte le funzionalità aggiuntive e non, del linguaggio e riportando le problematiche affrontate, esso avrà buone possibilità di progredire al meglio; tutt'ora, con i vantaggi che dispone rispetto agli altri linguaggi e con la possibilità di girare su ogni computer che abbia una JVM, si consiglia vivamente il suo uso.
Adesso si aspetta impazientemente un uso del linguaggio nativo di Kotlin e sopratutto delle rivoluzionarie Coroutine.

Bibliografia

[1] Neil Smyth, *Kotlin / Android Studio 3.0 Development Essentials: Android 8 Edition*, Createspace Independent Pub, novembre 2017

[2] Antonio Leiva, *Kotlin for Android Developers: Learn Kotlin the easy way while developing an Android App*, CreateSpace Independent Publishing Platform, marzo 2016

[3] Kotlin Learn and Android
https://kotlinlang.org/docs/reference/android-overview.html

[4] Great Circle Distance
https://en.wikipedia.org/wiki/Great-circle_distance

[5] Algoritmo Great Circle Distance, scritto in Java, per l'ordinamento dei luoghi

[6] Problema del commesso viaggiatore
https://it.wikipedia.org/wiki/Problema_del_commesso_viaggiatore

[7] Ottenimento API KEY di Google Maps
https://developers.google.com/maps/documentation/android-sdk/signup

[8] Utilizzo di Retrofit in Kotlin
https://virtuooza.com/android-api-client-using-retrofit-rxjava-kotlin

[9] Kotlin-Android: Pro e Contro
https://medium.com/keepsafe-engineering/kotlin-the-good-the-bad-and-the-ugly-bf5f09b87e6f

[10] Creazione di una recycler view in Kotlin
https://medium.com/@paul.allies/kotlin-with-recyclerview-1637145b170f