Università degli Studi di Firenze

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di laurea in informatica

Tesi di Laurea Triennale in Informatica

Studio del linguaggio Kotlin per applicazioni Android

Candidato
Ferruzzi Gianmarco

Relatore
Prof. Bettini Lorenzo

Anno Accademico 2019 - 2020

A coloro con i quali avrei voluto condividere questo traguardo.

Indice

Elenco delle figure

Listings

Introduzione

1.1 Linguaggio Kotlin

Kotlin è un linguaggio open-source, fortemente tipizzato e orientato alla programmazione a oggetti, sviluppato dall'azienda di software JetBrains, a partire dal 2011. Il principale obiettivo del linguaggio è quello di, basandosi sulla *Java Virtual Machine*, interoperare con Java Runtime Envirorment, rendendolo pienamente compatibile con l'ecosistema Java, in modo da poter sfruttare tutti gli strumenti già disponibili. Kotlin propone una soluzione coincisa a molti problemi presenti in Java, come il problema del puntatore nullo a runtime, che verrà trattato in seguito. Ad oggi Kotlin viene usato in vari ambiti, come Web Frontend, applicazioni Server-Side e, come nel caso di questo progetto, per lo sviluppo di applicazioni mobile Android.

1.2 Scopo della tesi

Lo scopo di questa tesi è lo studio del linguaggio Kotlin, sia negli aspetti comuni con il linguaggio Java sia nelle differenze fra i due. Fra queste differenze, sono state approfondite le differenze riguardanti la nullability nei due linguaggi, come viene gestita la programmazione funzionale e, infine, le differenze nella gestione della concorrenza.

Dopo una fase iniziale di studio del linguaggio l'obiettivo era anche quello di sviluppare una applicazione mobile in ambiente Android dove andare ad applicare le nuove conoscenze acquisite.

1.3 Struttura della tesi

La composizione di questo documento seguirà la seguente organizzazione:

- **Kotlin**: descrizione del linguaggio[1], con particolare attenzione alle differenze citate in precedenza
- **Android**: breve introduzione alla programmazione Android
- **Sviluppo dell'applicazione**: progettazione e implementazione dell'applicazione
- **Conclusioni**: valutazione del linguaggio e dello sviluppo dell'applicazione

[1]Nel capitolo dedicato alla descrizione del linguaggio sono state omesse le spiegazioni teoriche sulla programmazione.

Capitolo 2

Kotlin

In questo capitolo verrà descritto il linguaggio Kotlin, ponendo particolare attenzione alle differenze tra questo e il linguaggio Java.

2.1 Introduzione al linguaggio

2.1.1 Storia

Nel Luglio 2011, l'azienda *JetBrains* rivela il progetto Kotlin, un nuovo linguaggio per la *Java Virtual Machine*. Dopo 5 anni di sviluppo, questo linguaggio viene rilasciato ufficialmente nel Febbraio 2016. Nel 2017 Google annuncia il pieno supporto di Kotlin per Android e dal 2019 ne diventa il linguaggio di default per lo sviluppo di applicazioni.

2.1.2 Caratteristiche

Kotlin è un linguaggio molto flessibile in quanto possiede sia i costrutti per la programmazione *object-oriented* che quelli per la programmazione funzionale, i quali possono essere usati in concomitanza nello sviluppo di ap-

plicazioni. Le applicazioni che possono essere sviluppate in Kotlin sono di vari tipi e per diverse piattaforme, grazie alla possibilità di una programmazione multipiattaforma. Infatti l'uso del linguaggio Kotlin può essere fatto per applicazioni Android, server-side, client-side web e native. Tutto questo si basa sulla piena compatibilità con la Java Virtual Machine e la garanzia di poter utilizzare tutte le librerie già sviluppate per Java. Grazie a questa dualità fra Kotlin e Java viene offerta la possibilità di poter convertire facilmente qualunque progetto Java in Kotlin, anche grazie all'uso di un tool messo a disposizione degli sviluppatori per permettere una traduzione istantanea del codice.
Kotlin è totalmente open-source; il suo codice sorgente è ottenibile da chiunque dalla piattaforma GitHub in modo gratuito.

2.2 Sintassi di base

In questa sezione verrà analizzata brevemente la sintassi di base del linguaggio.

2.2.1 Variabili

In Kotlin è possibile definire due tipi di variabili:

- **var**: queste sono variabili il cui valore può variare nel tempo.
- **val**: queste variabili sono invece *read-only*, ovvero una volta assegnate non possono essere assegnate di nuovo. Queste coincidono con le variabili dichiarate `final` in Java.

Tutte le varibili sono oggetti, in quanto non esistono tipi primitivi, che si trovano in Java, e non esiste il tipo `void`. Se una funzione non ritorna niente,

in realtà questa ritorna un oggetto di tipo `Unit`, anche se non esplicitamente definito.

Ogni variabile in Kotlin ha un tipo ben definito, che può essere ricavato dall'assegnazione.

Listing 2.1: Dichiarazione di variabili

```
1 //dichiariazione esplicita del tipo della variabile
2 var variabile1: tipoVariabile = valoreVariabile
3
4 //dichiariazione implicita del tipo della variabile
5 val variabile2 = "nuova variabile"
```

Il compilatore di Kotlin inoltre effettua il *type casting* in maniera automatica se ha la possibilità di capire automaticamente il tipo della variabile dopo il casting.

Listing 2.2: Type casting automatico

```
1 if (x is String) {
2     print(x.length) // x viene automaticamente castata a String
3 }
```

Vettori

In Kotlin i vettori sono **invarianti**. Ciò significa, ad esempio, che non è possibile assegnare esplicitamente un `Array<String>` ad un `Array<Any>`, in modo da prevenire possibili errori a runtime.

2.2.2 Tipi base

In Kotlin ci sono tre tipi base: `Any`, `Unit`, `Nothing`.

Any

Così come *Object* rappresenta la radice della gerarchia in Java, *Any* rappresenta il supertipo di tutti i tipi ***non-nullable***. Ci sono due principali differenze tra Any e Object: In Java, i tipi primitivi non fanno parte della gerarchia dei tipi e necessitano di essere wrappati da altri tipi, mentre in Kotlin Any è il super tipo di tutti i tipi; Any non può contenere `null`, nel qual caso viene usato il tipo `Any?`. Any ha tre metodi dichiarati, che quindi sono definiti per ogni classe Kotlin: `equals()`, `hashCode()` e `toString()`.

Unit

In Java, per indicare che una funzione non ritorna niente viene usato il tipo `void`, *Unit* è il suo equivalente in Kotlin. A differenza di `void` però, `Unit` è un tipo effettivo, quindi può essere passato come argomento ad una funzione.

Nothing

Questo tipo non esiste in Java. Viene usato quando una funzione non terminerà mai, normalmente, e quindi un valore di ritorno non avrebbe senso. Diventa molto utile per capire quando una funzione non termina mai, ad esempio in funzioni come il ciclo principale di un engine di gioco.

2.2.3 Modificatori

In Kotlin si hanno gli stessi modificatori che si trovano in Java con l'aggiunta del modificatore `internal`, che da visibilità solo all'interno dello stesso modulo. A differenza di Java, il modificatore di default è il modificatore `public`.

2.2.4 Espressioni condizionali

In aggiunta alle espressioni condizionali presenti in Java, come l'istruzione `if` oppure l'istruzione `for`, in Kotlin è presente l'istruzione `when`, che permette di eseguire parti di codice in base al valore della variabile che viene passata, in maniera simile ad uno *switch* in Java.

Listing 2.3: Istruzione when

```
1 when(numero) {
2     1 -> println(1)
3     2 -> println(2)
4     3 -> println(3)
5     else -> println("Il valore del numero non risulta compreso tra 1 e 3")
6 }
```

Il codice che verrà eseguito dipenderà solo dal valore della variabile `numero`, e nel caso in cui non ci siano corrispondenze, verrà eseguito il codice presente nel blocco `else`. Tramite l'utilizzo della keyword *in*, il blocco di codice precendente può essere scritto in modo più compatto.

Listing 2.4: Istruzione when con in

```
1 when(numero) {
2     in 1..3 -> println(numero)
3     else -> println("Il valore del numero non risulta compreso tra 1 e 3")
4 }
```

2.2.5 Collezioni

Le collezioni presenti nelle librerie base di Kotlin sono le stesse presenti in Java, tutte appartenenti alla classe `Collection`, ma in questo linguaggio ogni collezione ha la relativa collezione mutabile. In Kotlin infatti ci sono due tipi di collezioni:

- *mutabili*: queste collezioni permettono di essere modificate, aggiungendo o rimuovendo elementi da esse.

- *non mutabili*: queste collezione invece permettono solo le operazioni di lettura, quindi è possibile definirle collezioni *read-only*.

Listing 2.5: Tipi di collezioni in Kotlin

```
1 val nomiCittaMutabili: MutableList<String> = mutableListOf("Firenze")
2 val nomiCittaImmutabili: List<String> = nomiCittaMutabili
3 nomiCittaMutabili.add("Roma") //aggiunge "Roma" alla lista mutabile
4 //nomiCittaImmutabili.add("Roma") non risulta invece permesso
5 nomiCittaMutabili.clear()
```

2.2.6 Eccezioni

Kotlin non modifica la gestione delle eccezioni di Java. In Kotlin però non esistono le eccezioni controllate, infatti per il compilatore non è obbligatorio effettuare un `catch` per nessuna eccezione.

2.2.7 Funzioni

In Kotlin, per dichiarare una funzione viene utilizzata la parola chiave `fun`, seguita dal nome della funzione, dai parametri (i quali devono essere ben tipizzati) e infine dal tipo di ritorno di tale funzione. Nel caso in cui la funzione non abbia valori di ritorno, viene usato il tipo *Unit*, il quale può essere specificato oppure omesso.

Listing 2.6: Funzione

```
1 fun somma(param1: Int, param2: Int): Int{...}
2 fun stampaValore(param1: Int): Unit{...} //Unit esplicito
3 fun stampaValore(param1: Int){...} //Unit implicito
```

I parametri di una funzione posso avere dei valori di default, i quali vengono usati nel caso in cui alla funzione non siano passati i valori associati a tali parametri.

Listing 2.7: Funzione con parametri default

```
fun radice(valore1: Int,
        esponente: Int = 2 //esponente ha 2 come valore di default
        ): Int{...}
```

Se viene eseguito l'override di una funzione con valori di default è possibile cambiare o rimuovere il valore di default.

Se il parametro con valore di default è seguito da un parametro non avente un valore di default, allora è possibile utilizzarlo solo tramite l'uso dei *parametri nominati*. Questo meccanismo permette di specificare il valore da passare allo specifico parametro, potendo quindi anche passare i parametri in ordine differente da quello definito nella funzione.

Listing 2.8: Funzione con parametri nominati

```
val risultato = radice(esponente = 3, valore1 = 10)
```

Funzioni inline e infix

Kotlin permette l'utilizzo delle funzioni *inline*, ovvero permette di scrivere funzioni le quali contengono una singola espressione in modo compatto, evitando di usare le parentesi graffe, usando solo una riga di codice.

Listing 2.9: Funzione inline

```
fun raddoppia(x: Int): Int = x * 2
```

è possibile inoltre definire delle funzioni le quali posso essere richiamate senza l'uso del punto e delle parentesi per i parametri di input, tramite l'utilizzo della parola chiave `infix`.

Listing 2.10: Funzione infix

```
infix fun Int.somma(x: Int): Int { ... }

val x = 1 somma 3 //notazione infix
val x = 1.somma(3) //notazione comune
```

Funzioni locali e globali

Kotlin supporta le funzioni locali, ovvero funzioni definite all'interno di altre funzioni.

Listing 2.11: Funzione interna

```
1  fun accumula(valore: Int): Int {
2      var valoreTotale = valore
3      fun aggiungi() { //funzione interna
4          valoreTotale++
5      }
6      for (i in 1..10) {
7          aggiungi()
8      }
9      return valoreTotale
10 }
```

Inoltre è possibile dichiarare delle funzioni globali, ovvero accessibili ovunque, senza che esse siano contenute in una classe. La dichiarazione di tale tipo di funzioni viene fatta in un file apposito.

2.3 Programmazione ad oggetti

Kotlin, come ovviamente anche Java, implementa il paradigma della programmazione ad oggetti, tramite la possibilità di definire degli oggetti e farli interagire tra loro. In questa sezione verrà descritto come Kotlin adotta questo paradigma e come si differenzia da Java nel farlo.

2.3.1 Classe

La dichiarazione di una classe in Kotlin non si discosta molto dalla stessa dichiarazione in Java. La parola chiave per la sua dichiarazione è `class`, come in Java.

Listing 2.12: Dichiarazione di una classe

```
class Account {
    private var nome: String? = null
    private var saldo: Float = 0f
    private var acc_no: Int = 0

    fun deposito() {...}
    fun prelevo() {...}
    fun bilancio() {...}
}
```

Costruttori

Una classe in Kotlin può avere un *costruttore primario* ed uno o più *costruttori secondari*. Il costruttore primario viene scritto subito dopo il nome della classe.

Listing 2.13: Costruttore primario

```
class Studente constructor(nome: String){
    /*..corpo della classe..*/
}
```

Se il costruttore primario non ha annotazioni o modificatori di visibilità (ad esempio se fosse necessario avere il costruttore primario privato), è possibile omettere la parola chiave `constructor`.

Listing 2.14: Costruttore primario con omissione di constructor

```
class Studente(nome: String){
    /*..corpo della classe..*/
}
```

Il costruttore primario non può contenere codice. Il codice necessario la inizializzazione può essere inserito nei blocchi di inizializzazione, identificato dalla parola chiave `init`. Possono essere presenti più blocchi di inizializzazione, i quali saranno eseguiti nell'ordine in cui appaiono.

Listing 2.15: Blocco init

```
class Studente(nome: String){
    private val identificatore: String
    init{
        identificatore = nome + "_id"
        println(identificatore)
    }
}
```

Possono essere presenti più costruttori secondari, i quali devono richiamare quello primario tramite la parola chiave `this`.

Listing 2.16: Blocco init

```
class Studente(nome: String){
    private val identificatore: String

    constructor(nome: String, cognome: String): this(nome){
        //codice costruttore secondario
    }

    init{
        identificatore = nome + "_id"
        println(identificatore)
    }
}
```

Data Class

Se lo scopo di una classe è solo quello di mantenere dei dati, Kotlin mette a disposizione le *data classes*, simili alle classi definite `record` in Java, con alcune aggiunte che ne facilitano l'utilizzo.

Listing 2.17: Data class

```
data class Studente(val nome: String, val cognome: String)
```

Per queste classi il compilatore genera automaticamente i costruttori ed alcune funzioni, derivandole dal contenuto della classe stessa, come i getter e

i setter. Tra queste funzioni è presente la funzione `copy()` che permette di copiare un oggetto (copia non in profondità) modificandone alcuni valori dei suoi attributi.

Listing 2.18: Funzione copy()

```
1 val jack = User(name = "Jack", age = 1)
2 val olderJack = jack.copy(age = 2)
```

La funzione `componentN()` permette di selezionare i componenti della classe. Questa funzione permette di usare le dichiarazioni destrutturate (*Destructuring declarations*), ovvero scomporre un oggetto nelle sue componenti, inserendole in delle variabili. Facendo riferimento alla classe dichiarata al punto 2.12, un esempio di dichiarazione destrutturata può essere:

Listing 2.19: Dichiarazione destrutturata

```
1 val account = Account(...)
2 val (nome, saldo) = account
```

Questa dichiarazione destrutturata viene compilata come:

Listing 2.20: Dichiarazione destrutturata compilata

```
1 val nome = account.component1()
2 val saldo = account.component2()
```

Le Data class per essere consistenti devono rispettare alcuni requisiti quali: 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.

La classi `record` in Java seguono lo stesso principio delle classi `data`, in quanto anche queste permettono di mantenere dei dati e forniscono alcune funzioni di default, come le funzioni `equals()`, `hashCode()` e `toString()` alle quali Kotlin aggiunge anche la funzione `copy()` descritta in precedenza. In aggiunta a questa è possibile identificare ulteriori differenze sostanziali, quali:

- le classi `record` non permettono la modifica dei loro attributi in quanto questi sono tutti definiti `private` e `final`; al contrario, Kotlin, permette di definire gli attributi `var`, permettendone quindi una possibile modifica.
- le classi `data` possono estendere altre classi (2.3.2), al contrario delle classi `record`, le quali non possono farlo direttamente.
- le classi `data` permettono, a differenza delle classi `record`, di dichiare delle variabili di istanza al loro interno.

Companion Object

A differenza di Java, Kotlin non permette variabili o funzioni statiche. Se è necessario scrivere una funzione la quale possa essere chiamata senza avere l'istanza della relativa classe ma necessita dell'accesso ai membri di tale classe, allora è possibile dichiararla come membro del *companion object* interno alla classe. Un particolare utilizzo si può trovare quando è necessario implementare il pattern *Singleton*.

Listing 2.21: Companion Object

```
1 class EventManager {
2     companion object FirebaseManager {
3         /*..codice del companion object..*/
4     }
5 }
6
7 val firebaseManager = EventManager.FirebaseManager
```

Inline classes

A volte può essere necessario creare dei wrapper per alcuni tipi di variabili, e spesso questo porta allo spreco di risorse, specialmente se i wrapper sono

creati per tipi primitivi, i quali sono ottimizzati e venendo inseriti in questi wrapper perdono la loro ottimizzazione. Per ovviare a questo problema Kotlin ha introdotto, anche se al momento della stesura di questo documento sono ancora in fase *Beta*, le *inline classes*. Queste classi sono un sottoinsieme delle *Data class*, non hanno una vera identità e possono solo contenere dati. La dichiarazione di una classe *inline* è fatta inserendo la parola chiave `inline` o `value` prima del nome della classe.

Listing 2.22: Dichiarazione classe inline

```
1 inline class Password(val value: String)
2
3 value class Password(private val s: String)
```

Listing 2.23: Uso delle classe inline

```
1 // Non vi e' una vera istanziazione della classe Password
2 // A runtime 'securePassword' conterra' solo 'String'
3 val securePassword = Password("La mia password")
```

Oggetti

Per instanziare un oggetto in Kotlin, a differenza di Java, non è necessario l'uso della parola chiave `new`.

Listing 2.24: Istanziazione

```
1 val securePassword = Password("La mia password")
```

2.3.2 Ereditarietà

Tutte le classi in Kotlin hanno la stessa superclasse *Any* (2.2.2), la quale è anche la superclasse di default per le classi che non hanno superclassi dichiarate esplicitamente. Di default le classi Kotlin non posso essere ereditate. Per renderlo possibile è necessaria la parola chiave `open`. Per indicare

che una classe ne estende un'altra, viene indicato il nome della classe estesa, preceduto da `:`, dopo il nome (e il costruttore) della classe che la estende.

Listing 2.25: Classi open ed ereditarietà

```
1 open class Persona(nome: String)
2
3 class Studente(nome: String, matricola: String): Persona(nome)
```

Override

Come per le classi, anche i metodi, per poterne effettuare l'override, devono essere *open*. Se un metodo non ha il modificatore *open*, dichiarare un metodo con la stessa firma, con o senza la parola chiave `override`, in una sottoclasse non è ammesso. Un metodo con la parola chiave `override`, diventa open per le sottoclassi. Per non permetterlo è necessario aggiungere la parola chiave `final`.

Listing 2.26: Override dei metodi

```
1  open class Figura {
2      open fun disegna() { /*...*/ }
3      fun riempi() { /*...*/ }
4  }
5
6  class Cerchio() : Figura() {
7      override fun disegna() { /*...*/ }
8      //override fun riempi() non e' permesso
9  }
10
11 open class Rettangolo() : Figura() {
12     final override fun disegna() { /*...*/ }
13 }
```

In Kotlin è anche possibile eseguire l'override degli attributi delle classi, seguendo lo stesso principio dei metodi. Ad esempio è possibile eseguire l'override di un attributo definito *val*, rendendolo *var*.

Classi astratte

Le classi astratte di Kotlin seguono lo stesso principio di Java, quindi possono contenere metodi non implementati. Un classe o un metodo astratto non necessita di essere dichiarato *open*, il quale però può essere usato per permettere l'override di un metodo non astratto presente nella classe astratta.

Listing 2.27: Classe astratta

```
1  open class Poligono {
2      open fun disegna() {...}
3  }
4  abstract class Rettangolo : Poligono() {
5      abstract override fun disegna()
6  }
```

Interfacce

Come in Java, anche in Kotlin non è possibile avere una classe che estende più di una classe. Viene adottato per questo il concetto di interfaccia. Una classe Kotlin può implementare una interfaccia o più mentre una interfaccia può estendere una o più interfacce. Le interfacce possono contenere dichiarazioni di metodi, che possono anche essere astratti, e di attributi che posso anch'essi essere astratti.

Listing 2.28: Interfacce

```
1  interface InterfacciaPadre {
2      val proprietaAstratta: Int
3
4      val proprietaConImplementazione: String
5          get() = "proprieta' implementata"
6
7      fun stampaProprietaAstratta() {
8          print(proprietaAstratta)
9      }
10
11     fun stampaProprietaImplementata()
```

```
12 }
13
14 class Figlio : InterfacciaPadre {
15     override val proprietaAstratta: Int = 10
16     override fun stampaProprietaImplementata(){...}
17 }
```

Estensioni

Kotlin permette di estendere le classi, aggiungengo delle funzionalità, senza nè utilizzare l'ereditarietà nè modificandole, tramite le *extensions*. I metodi aggiunti in questo modo sono accessibili come fossero metodi della classe originale. L'estensione viene risolta staticamente e non viene fatta alcuna valutazione a tempo di esecuzione.

Listing 2.29: Estensione di una classe

```
1 fun Classe.nuovoMetodo(){
2     /*...*/
3 }
```

2.4 Programmazione Funzionale

2.4.1 Funzioni High-Order e Lambda

In Kotlin le funzioni possono essere salvate in una variabile, passate come argomento e ritornate da altre funzioni di ordine superiore. Una funzione di ordine superiore (*High-Order*) è una funzione che ha come parametro di ingresso o di uscita una funzione. Le funzioni di questo tipo adottano una notazione specifica per indicare i parametri di ingresso e di uscita, ovvero `(A , B) -> C`. Questa notazione indica il tipo dei parametri di ingresso, `A` e `B`, ed il tipo del parametro di uscita, `C`. In questo tipo di notazione, in caso di tipo di ritorno `Unit`, questo non è omissibile

Listing 2.30: Funzione High-Order con parametro una funzione

```
fun calcola(x : Int, y: Int, operazione: (Int, Int) -> Int): Int{
    return operazione(x,y)
}

fun somma(x: Int, y: Int) = x + y

fun main(){
    //tramite ::somma viene richiamata la funzione somma
    val risultatoSomma = calculate(4, 5, ::somma)
    val risultatoMoltiplicazione = calculate(4, 5, {a, b -> a * b})
}
```

Listing 2.31: Funzione High-Order che ritorna una funzione lambda

```
fun operazione(): (Int) -> Int {
    return {x -> x * 2}
}

fun main(){
    val funct = operazione()
    println(funct(2))
}
```

2.4.2 Funzioni let, run, also, apply, with

La libreria standard di Kotlin, *kotlin-stdlib*, fornisce alcune importanti funzioni high-order.

Let

La funzione *let* prende l'oggetto sulla quale è invocata come input ed esegue l'espressione lambda che viene definita all'interno di essa.

Listing 2.32: Istruzione Let

```
fun main(args: Array<String>) {
    var str = "Hello World"
    str.let{ println("$it!!") } //stampa "Hello World!!"
```

```
4     println(str) //stampa "Hello World"
5 }
```

Questa funzione può essere utilizzata per verificare che una variabile non sia nulla. Questa funzione, infatti, viene eseguita solo se il contenuto della variabile non è nullo.

Listing 2.33: Istruzione Let per null checks

```
1 fun main(args: Array<String>) {
2     var name : String? = "Kotlin let null check"
3     name?.let { println(it) } //stampa "Kotlin let null check"
4     name = null
5     name?.let { println(it) } //non accade niente
6 }
```

Run

A differenza della funzione *let*, la funzione *run* permette di ritornare il risultato dell'espressione definita al suo interno e di modificare il valore della variabile esterna al blocco stesso.

Listing 2.34: Istruzione Run

```
1 fun main(args: Array<String>) {
2     var esempio = "Esempio Run"
3     println(esempio) //stampa "Esempio Run"
4     esempio = run {
5         val esempio = "Hello World!"
6         esempio // ritorna la variabile esempio
7     }
8     println(esempio) //stampa "Hello World!"
9 }
```

Also

La funzione *also* permette di eseguire della logica aggiuntiva all'oggetto sul quale viene invocata. La funzione *also* però ritorna sempre l'oggetto

originale, immutato, invece di nuovi dati, come *run* o *let*.

Listing 2.35: Istruzione Also

```
fun main(args: Array<String>) {
    var m = 3
    m = m.also {println(it + 2)} //stampa 5
    println(m) //stampa 3
}
```

Apply

La funzione *apply* viene eseguita su oggetti e ritorna l'oggetto dopo che viene eseguito su di esso l'espressione che questa contiene. Questa funzione utilizza `this` per fare riferimento all'oggetto sul quale viene eseguita.

Listing 2.36: Istruzione Apply

```
data class Persona(var nome: String, var esempio : String)
fun main(args: Array<String>) {
    var dipendente = Persona("Marco", "Kotlin")
    dipendente.apply { this.esempio = "Java" }
    println(dipendente) //stampa Persona("Marco", "Java")
}
```

With

La funzione *with* segue lo stesso principio della funzione *apply*, ma permette di essere eseguita con una sintassi differente.

Listing 2.37: Istruzione With

```
data class Persona(var name: String, var esempio : String)
fun main(args: Array<String>) {
    var gestore = Persona("Luigi", "Java")
    with(gestore){
        nome = "Mario"
        esempio = "Kotlin"
    }
}
```

2.5 Nullability del linguaggio

Una delle più comuni insidie in molti linguaggi di programmazione, incluso Java, risiede nella possibilità di accedere a riferimenti nulli, portando ad una *NullPointerException* (NPE). Il sistema dei tipi di Kotlin mira ad eliminare le *NullPointerExceptions* dal codice. Infatti si hanno solo pochi casi in cui in Kotlin c'è la possibilità che avvenga una *NPE*, quali:

- una chiamata esplicita a `throw NullPointerException()`
- uso dell'operatore `!!`, che verrà in seguito descritto
- possibile inconsistenza durante l'inizializzazione
- interoperazione con Java

Il sistema dei tipi di Kotlin distingue i riferimenti che posso contenere valori nulli da quelli che non possono.

Listing 2.38: Riferimenti nulli non permessi

```
1 var a: String = "abc" // normale inizializzazione, non-null di default
2 a = null // errore di compilazione
```

Per permettere ad una variabile di contenere un riferimento nullo, essa va dichiarata come *nullable*, aggiungendo `?` nel tipo della variabile. Per `String`, ad esempio, si ottiene `String?`.

Listing 2.39: Riferimenti nulli permessi

```
1 var b: String? = "abc" // permette di impostarlo a null
2 b = null // ok
3 print(b)
```

2.5.1 Controlli

Grazie al fatto che è possibile conoscere a priori se una variabile può contenere o meno valori nulli, i controlli da effettuare per evitare errori si riducono alle sole variabili dichiarate *nullable*.
Per effettuare questi controlli Kotlin permette varie modalità.

Controllo Esplicito

Il controllo più immediato risulta quello di andare a controllare esplicitamente se la variabile contiene un valore nullo, e gestire le due opzioni in modo separato. Un esempio semplice di questa strategia si trova nell'uso del costrutto `if`.

Listing 2.40: Controllo esplicito

```
1 val l = if (b != null) b.length else -1
```

Al posto del costrutto `if` è possibile usare l'operatore *Elvis* `?:`, il quale permette di scrivere l'espressione precedente in maniera più compatta.

Listing 2.41: Controllo esplicito con operatore Elvis

```
1 val l = b?.length ?: -1
```

Safe Calls

Una seconda opzione risiede nell'uso delle *safe calls*(chiamate sicure), tramite l'operatore delle *safe calls*, `?.`. Di seguito un esempio di uso di tale operatore.

Listing 2.42: Safe Calls

```
1 val a = "Kotlin"
2 val b: String? = null
3 println(b?.length)
4 println(a?.length) // safe call non necessaria
```

In questo esempio, `b?.length` restituisce `b.length` se `b` non è nullo, `null` altrimenti.

Le chiamate sicure sono utili quando è necessario concatenare varie chiamate su un oggetto. Ad esempio, se un dipendente è assegnato ad un dipartimento e fosse necessario ottenere il nome del caporeparto, è possibile scrivere l'espressione per ottenere tale nome come:

Listing 2.43: Esempio di catena di chiamate sicure

```
1 val nomeCaporeparto = dipendente1?.dipartimento?.caporeparto?.nome
2 // nomeCaporeparto sara' di tipo "String?"
```

Questa catena di chiamate sicure restituisce `null` se una qualsiasi delle sue componenti è nulla. Per eseguire una operazione solo per i valori non nulli, è possibile utilizzare l'operatore `?.` insieme alla funzione `let` (2.4.2), la quale viene eseguita solo se il contenuto della variabile non è nullo.

Listing 2.44: Esempio di funzione let con chiamata sicura

```
1 val listWithNulls: List<String?> = listOf("Kotlin", null)
2 for (item in listWithNulls) {
3     item?.let { println(it) } // stampa Kotlin e ignora null
4 }
```

Safe Casts

Il casting sicuro evita le eccezioni di tipo *ClassCastException*, le quali avvengono se il tipo verso il quale sta venendo effettuato il cast non corrisponde al tipo della variabile che sta subendo il casting, permettendo di far ritornare `null` al casting.

Listing 2.45: Safe Cast

```
1 val a = "Kotlin"
2 val aInt: Int? = a as? Int //aInt conterra' il valore null
```

Operatore !!

L'operatore `!!` permette di fare asserzioni sul valore contenuto in una variabile, asserendo che tale valore non sia nullo. Questo operatore infatti converte qualsiasi tipo di un valore in un tipo non nullo, e lancia una NPE se il valore è nullo.

Listing 2.46: Operatore !!

```
1 val b: String? = "Kotlin"
2 val bNotNull = b!! // bNotNull sara' di tipo String
3 b = null
4 val bNull = b!! // sollevera' una eccezione
```

2.6 Coroutines

In una applicazione vi è molto spesso la necessità di eseguire funzioni che richiedono molto tempo per terminare (come una chiamata ad un servizio web), le quali, per evitare il blocco dell'applicazione stessa, devono essere eseguite in modo asincrono. In Java questo processo viene gestito dai *Threads*. L'uso di molti threads, anche se permette di eseguire più compiti in contemporanea, va a peggiorare le performance dell'applicazione, in quanto i thread hanno un costo, in termini di risorse della cpu, dovuto alla loro allocazione e al loro *context-switch*, che, all'aumentare del numero dei thread presenti, fa diminuire le performance dell'applicazione.
Per ovviare a questo problema, Kotlin ha introdotto le *Coroutines*, simili ai thread in Java, ma con un costo computazionale minore.
Le Coroutines hanno un costo computazionale minore in quanto, mentre i thread creati in Java sono associati a thread di sistema specifici, queste vengono eseguite da *pool di background thread* e non sono quindi associate a nessun thread in particolare. Ciò permette ad una coroutine di essere avviata in un

thread, essere sospesa e poi riavviata su un altro thread.
Le Coroutines non sono gestite dal sistema operativo, ma dal *Kotlin Runtime*.

2.6.1 Funzioni Suspending

Le funzioni *suspending* sono al centro del concetto di coroutines. Una funzione *suspending* è semplicemente una funzione la cui esecuzione può essere messa in pausa e fatta ripartire un altro momento. Possono eseguire lunghe operazioni e aspettare che queste si concludino senza bloccare l'applicazione. La sintassi di una funzione di questo tipo risulta molto simile ad una qualsiasi funzione in Kotlin, con però l'aggiunta della parola chiave `suspend` prima della dichiarazione della funzione stessa.
Le funzioni *suspending* possono essere invocate solo da altre funzioni *suspending* o da coroutines.

Listing 2.47: Funzione suspending

```
1 suspend fun backgroundTask(param: Int): Int {
2     // operazione che richiede molto tempo
3 }
```

A runtime, una funzione suspending viene convertita dal compilatore in un'altra funzione che non contiene `suspend`, la quale avrà un parametro di ingresso aggiuntivo di tipo `Continuation<T>`. L'esempio di funzione precedente verrà quindi convertito in:

Listing 2.48: Funzione suspending convertita a runtime

```
1 fun backgroundTask(param: Int, callback: Continuation<Int>): Int {
2    //  operazione che richiede molto tempo
3 }
```

`Continuation<T>` è una interfaccia che contiene due funzioni, le quali sono invocate per far ripartire la coroutine, con un valore di ritorno o con un'eccezione se si è verificato un errore durante la sospensione della funzione.

Listing 2.49: Interfaccia Continuation<T>

```
1 interface Continuation<in T> {
2     val context: CoroutineContext
3     fun resume(value: T)
4     fun resumeWithException(exception: Throwable)
5 }
```

2.6.2 Contesto delle Coroutines

Le Coroutines sono sempre eseguite in un certo contesto, contenente vari elementi. I due principali sono:

- **Job**, il quale concettualmente rappresenta un oggetto cancellabile avente un ciclo di vita ben definito.
- **Dispatcher**, che determina quale o quali thread la coroutine userà per le sua esecuzione.

Tramite il *dispatcher* è possibile confinare l'esecuzione di una coroutine ad uno specifico thread, ad una pool di thread oppure non confinarla affatto.

2.6.3 Funzioni principali

Funzione launch

La funzione `launch` avvia una nuova coroutine senza bloccare il thread corrente e ritorna un riferimento a tale coroutine come un oggetto di tipo `Job`. Questa funzione ha due parametri opzionali:

- *context*, ovvero il contesto nel quale la coroutine verrà eseguita. Se non definito, verrà dedotto dal `CoroutineScope` nel quale la coroutine è stata lanciata.

- *start*, cioè le opzioni di avvio della coroutine, come quando essere avviata. Di default la coroutine viene avviata nel momento in cui viene creata.

Listing 2.50: Funzione launch nel GlobalScope

```
1 val job = GlobalScope.launch {
2     println("${Thread.currentThread()} avviato.")
3 }
```

Nell'esempio precedente, la coroutine viene avviata nel `GlobalScope`, nel quale vengono definite le coroutine *top-level*, globali. Nell'uso comune, lanciare coroutine nel `GlobalScope` è fortemente sconsigliato.
Invece di avviare una coroutine nel `GlobalScope`, come avviene per i comuni thread (i thread sono sempre globali), e possibile lanciare una coroutine in uno *scope* specifico: quello nel quale l'operazione deve essere eseguita.
Nell'esempio successivo, la coroutine viene avviata all'interno del blocco `runBlocking` [1] senza specificare il contesto, il quale verrà ottenuto dal contesto del blocco `runBlocking`.

Listing 2.51: Funzione launch

```
1 runBlocking {
2     val job = launch {
3         println("${Thread.currentThread()} avviato.")
4     }
5 }
```

[1] Un blocco `runBlocking` avvia una nuova coroutine e blocca il thread corrente fino a che questa non finisce.

Funzione async

La funzione `async` crea una coroutine e ritorna un *future*[2] del suo risultato come una istanza di `Deferred<T>`. *Deferred* è un *future* cancellabile non bloccante.

Listing 2.52: Funzione async

```
1 val deferred = async {
2     return@async "${Thread.currentThread()} avviato."
3 }
```

Come con la funzione `launch` (2.6.3), è possbile specificare un contesto nel quale eseguire la coroutine, e assegnare un valore al parametro *start*.

Listing 2.53: Funzione async con contesto

```
1 val deferred = async(Dispatchers.Unconfined, CoroutineStart.LAZY) {
2     println("${Thread.currentThread()} has run.")
3 }
```

2.7 Interoperabilità con Java

In questa ultima sezione saranno discussi alcuni dei metodi con i quali Kotlin e Java possono cooperare e coesistere.

2.7.1 Utilizzare Java da Kotlin

Kotlin è stato progettato pensando all'interoperabilità con Java. Il codice Java esistente può essere chiamato da Kotlin in modo naturale.

Listing 2.54: Utilizzo di classi Java in Kotlin

```
1 fun demo(source: List<Int>) {
2     val list = ArrayList<Int>()
```

[2]Il termine *future*, nella programmazione, viene utilizzato per indicare un oggetto che agisce come intermediario per il risultato della coroutine che è inizialmente sconosciuto

```
3
4       for (item in source) {
5           list.add(item)
6       }
7   }
```

Null Safety per Java

Qualsiasi riferimento in Java può essere `null`, il che rende impraticabili i requisiti di Kotlin per la *null safety*. I tipi delle dichiarazioni Java sono quindi trattati in modo specifico e chiamati **tipi di piattaforma**. I controlli di Kotlin per garantire la *null safety* su questi tipi non vengono applicati, rendendo possibili errori di tipo `NullPointerException` a runtime. È comunque possibile passare i valori di questi tipi ai tipi nullable di Kotlin in modo da essere gestiti come tali.

Mappatura di alcuni tipi

Alcuni tipi presenti in Java sono gestiti in modo specifico da Kotlin. Questi tipi di Java non sono semplicemente utilizzati, ma subiscono una mappatura su dei tipi specifici di Kotlin. La mappatura avviene solo in fase di complizione, mentre a runtime la loro rappresentazione non viene alterata. Un esempio di tipi che vengono mappati sono i tipi primiti di Java, che vengono appunto mappati nei corrispettivi tipi in Kotlin (`int` → `kotlin.Int`).

Eccezioni controllate

Dato che in Kotlin non esistono le eccezioni controllate (2.2.6), chiamando un metodo Java che dichiara una eccezione controllata, il compilatore non forzerà alcuna azione aggiuntiva.

2.7.2 Utilizzare Kotlin da Java

Il codice Kotlin può essere facilmente chiamato da codice Java. Ad esempio, una istanza di una classe Kotlin può essere creata e essere usata in un metodo Java. Nonostante questo è necessario fare attenzione ad alcune differenze tra Java e Kotlin che necessitano di attenzione quando avviene questa interazione.

Attributi di una classe Kotlin

Un'attributo di una classe Kotlin viene tradotto in Java come:

- un metodo *getter*, il cui nome viene ottenuto aggiungendo `get` davanti al nome dell'attributo;
- un metodo *setter*, solo se la variabile era di tipo `var`, il cui nome viene ottenuto aggiungendo `set` davanti al nome dell'attributo;
- una varibile privata con lo stesso nome dell'attributo;

Ad esempio, `var nome: String` viene compilato come:

Listing 2.55: Traduzione attributo Kotlin in Java

```
1 private String nome;
2
3 public String getNome() {
4     return nome;
5 }
6
7 public void setNome(String nome) {
8     this.nome = nome;
9 }
```

Se il nome dell'attributo inizia con `is`, allora viene applicata una regola di conversione differente: il nome del metodo *getter* avrà lo stesso nome dell'attributo, mentre il metodo *setter* avrà come nome il nome dell'attributo,dove

al posto di `is` si troverà `set`. Questa regola non si applica solo ad attributi di tipo `Boolean`.

Companion object

Una funzione definita all'interno di un blocco `companion object` viene convertita in una funzione statica se tale funzione è preceduta dalla annotazione `@JvmStatic`, altrimenti viene convertita in una normale funzione non statica. Nel seguente esempio, la funzione `callStatic()` verrà tradotta come funzione statita mentre la funzione `callNonStatic()` no.

Listing 2.56: Traduzione di companion object

```
1  class C {
2      companion object {
3          @JvmStatic fun callStatic() {}
4          fun callNonStatic() {}
5      }
6  }
7
8  C.callStatic(); // non genera errori
9  C.callNonStatic(); // errore: funzione non statica
```

Eccezioni controllate

Come discusso in precedenza, Kotlin non utilizza le eccezioni controllate (2.2.6), quindi normalmente una funzione Kotlin tradotta in Java non possiede l'espressione `throws Exception`. Quindi, ad esempio, se in Kotlin è presente una funzione come la seguente, in Java, inserendola dentro un blocco `try-catch`, il compilatore segnalerà un errore in quanto quella funzione non dichiara `throws IOException`

Listing 2.57: Esempio di funzione Kotlin con eccezione

```
1  fun scriviSuFile() {
2      /*...*/
```

```
3     throw IOException()
4 }
```

Per risolvere ciò, nella funzione scritta in Kotlin è necessario appore l'annotazione `@Throws(IOException::class)`.

Listing 2.58: Esempio di funzione Kotlin con annotazione @Throws

```
1 @Throws(IOException::class)
2 fun scriviSuFile() {
3     /*...*/
4     throw IOException()
5 }
```

Null Safety

Quando viene chiamata una funzione Kotlin da Java, non è possibile impedire che ad essa venga passato un valore nullo ad un parametro **non-null**. Per evitarlo Kotlin effettua dei controlli a runtime per tutte le funzioni `public` che si aspettano parametri non nulli. Facendo ciò viene subito generata una `NullPointerException` in caso di errore.

Capitolo 3

Android

In questo capitolo verranno descritte brevemente le parti essenziali della programmazione Android.

3.1 Android SDK

Il *Software Development Kit* di Android fornisce allo sviluppatore tutti gli strumenti necessari per sviluppare applicazioni da eseguire sul sistema operativo Android. Fra gli strumenti forniti dall'*SDK* si trovano *API*, librerie e strumenti di sviluppo utili a compilare, testare e debuggare applicazioni Android. L'SDK, tramite l'*SDK Manager*, permette di installare anche solo parti di esso, nel caso alcune funzionalità non siano necessarie al progetto.

3.2 Android Studio

Android Studio, annunciato da Google nel 2013, figlio di *IntelliJ IDEA*, un IDE sviluppato da *JetBrains*, ha preso il posto di Eclipse come ambiente di sviluppo più usato per lo sviluppo di applicazioni Android.

L'editor offre molte funzionalità che facilitano lo sviluppo di applicazioni, come un emulatore integrato in grado di emulare molti dispositivi Android, un analizzatore di performance delle applicazioni, il completamento intelligente del codice e la gestione del *version control* tramite GitHub.

3.3 Basi di sviluppo Android

In questa sezione verranno affrontati gli elementi base dello sviluppo di applicazioni Android.

3.3.1 Activity

Un'Activity si può identificare come una "pagina" dell'applicazione. Tutte le applicazioni Android contengono almeno un'Activity e tutte le Activity interagiscono con l'utente. Queste si occupano infatti di creare l'interfaccia grafica nella quale andare a inserire le componenti necessarie per tale interazione. Un'Activity si può presentare all'utente in varie forme, come una pagina full-screen, un popup oppure inserite in altre pagine. Tutto ciò viene definito nel layout collegato all'Activity stessa.

Listing 3.1: Esempio di Activity vuota in Kotlin

```
1 class EmptyActivity : AppCompatActivity() {
2     override fun onCreate(savedInstanceState: Bundle?) {
3         super.onCreate(savedInstanceState)
4
5         //Viene specificato il layout dell'Activity
6         setContentView(R.layout.activity_empty)
7     }
8 }
```

Tutte le Activity hanno lo stesso ciclo di vita, che inizia quando queste vengono lanciate e può terminare quando esse vengono chiuse oppure quando

l'applicazioni che le contiene viene chiusa. Di seguito una delle illustrazioni più note della programmazione Android.

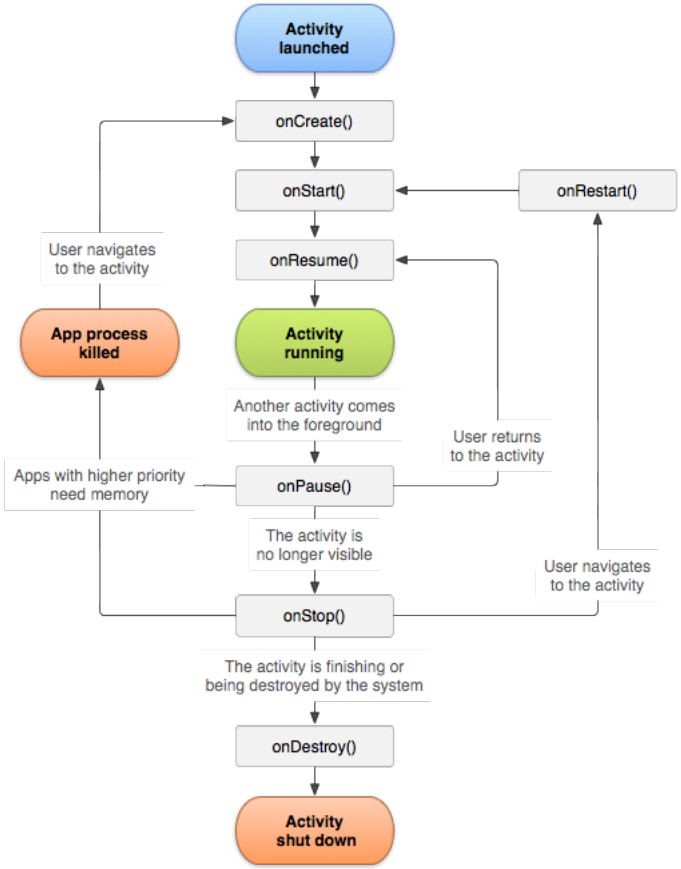

Figura 3.1: Il ciclo di vita di un'Activity

3.3.2 Layout

Per definire il layout di un'Activity è necessario creare un file appropriato, contenente le componenti grafiche e le relazioni che esistono tra esse. I file contenuti nella cartella *res/layout* contengono queste informazioni. Questi sono file di tipo *XML*, contenenti le gerarchie delle componenti e i loro attributi. Esistono vari tipi di layout, come il `LinearLayout`, quello più sem-

plice, che distribuisce in maniera sequenziale, seguendo l'ordinamento impostato, l'insieme di elementi che contiene. Ogni elemento contenuto in un layout ha un identificativo univoco, necessario per ottenerne un riferimento nell'Activity dove è inserito.

Listing 3.2: Esempio di file Layout per un'Activity

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".UI.EmptyActivity">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="EmptyActivity"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</LinearLayout>
```

3.3.3 Manifest

Ogni applicazione Android deve avere il file *AndroidManifest.xml* nella radice del progetto. Questo file descrive le informazioni essenziali dell'applicazione a tutte le entità che ne entreranno in contatto, come *Google Play* e il sistema operativo del dispositivo in cui verrà installata.
Tra le varie informazioni relative all'applicazione si trovano i permessi di cui questa necessita per funzionare correttamente, accedendo a parti protette del sistema o di altre applicazioni. Tra questi permessi si trovano quelli per

permettere l'accesso alla rete e per permettere l'accesso alla memoria interna del dispositivo.

3.3.4 Testing in Android

Nella programmazione Android, è necessario scrivere dei test sia per le singole classi e metodi sia per le interazioni con l'utente. Per i due tipi di test sono presenti due cartelle apposite nella struttura del progetto:

- ***tests***: questa cartella contiene i test che vengono eseguiti nella *Java Virtual Machine* locale, come i test JUnit
- ***androidTests***: questa cartella contiene i test che necessitano di essere eseguiti su dispositivi reali o virtuali, come quelli relativi all'interfaccia

Quando è necessario eseguire i test su di un dispositivo è possibile usare un device fisico, come il proprio cellulare Android, usare un device virtuale, come quello che mette a disposizione Android Studio oppure usare un device simulato, come quello di *Robolectric*.
I test che dovrebbero essere eseguiti per una applicazione Android si dividono principalmente in 3 categorie:

- **Small tests** (Unit Test), cioè quei test che servono per controllare la validità delle singole classi del progetto
- **Medium tests**, cioè quei test che servono per convalidare le interazioni tra i vari "livelli" logici dell'applicazione
- **Large tests**, cioè quei test per convalidare l'esperienza dell'utente sull'applicazione

La percentuale di presenza di questi tipi di test dovrebbe seguire la seguente piramide:

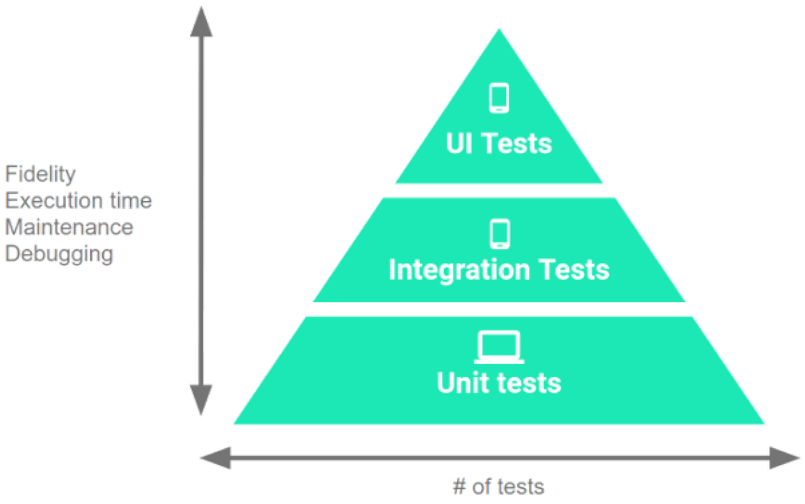

Figura 3.2: La Piramide del testing

Capitolo 4

Sviluppo dell'applicazione

4.1 Scopo dell'applicazione

Lo sviluppo di questa applicazione ha esclusivamente scopo di studio. L'obiettivo è quello di utilizzare le conoscenze acquisite nel corso dello studio del linguaggio Kotlin, in un progetto verosimile di applicazione Android. Per questo, è stato scelto di creare una applicazione progettualmente semplice.

4.2 Meteofy

Lo scopo dell'applicazione è quello di permettere all'utente utilizzatore di ricevere in tempo reale le condizioni metereologiche delle città inserite dall'utente stesso. Da qui deriva il nome che è stato dato all'applicazione, Meteofy, unione del termine "Meteo" e "Fy", diminutivo del nickname *Fyruz*, con cui il canditato firma i suoi progetti.

4.3 Progettazione

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

4.3.1 Interfaccia

L'interfaccia della applicazione rispecchia la necessità di dover avere la lista delle città selezionate dall'utente, visualizzabili in maniera immediata, con le condizioni metereologiche di tali città facilmente identificabili tramite una categorizzazione basata sui colori. Inoltre è presente la possibilità di eliminare una città semplicemente tenendo premuto sull'elemento contenente tale città.
Risulta chiaramente necessario avere un metodo che permetta all'utente di effettuare le ricerche, e quindi aggiungere, delle città. Deve essere, quindi, sempre presente un qualche pulsante, facilmente raggiungibile dall'utente, che renda possibile tale interazione.

4.3.2 Raccolta Dati

Per la raccolta dei dati metereologici in tempo reale vengono utilizzate delle *API* messe a disposizione gratuitamente dal servizio *OpenWeatherMap*, le quali vengono interrogate tramite chiamate *REST* effettuate direttamente dall'applicazione.
Il servizio di raccolta dati in tempo reale offerto da OpenWeatherMap risponde alle richieste inviate dall'applicazione con dati strutturati in formato JSON, i quali dovranno poi venire tradotti in modo da poterli gestire e salvare con facilità nel database interno all'applicazione.

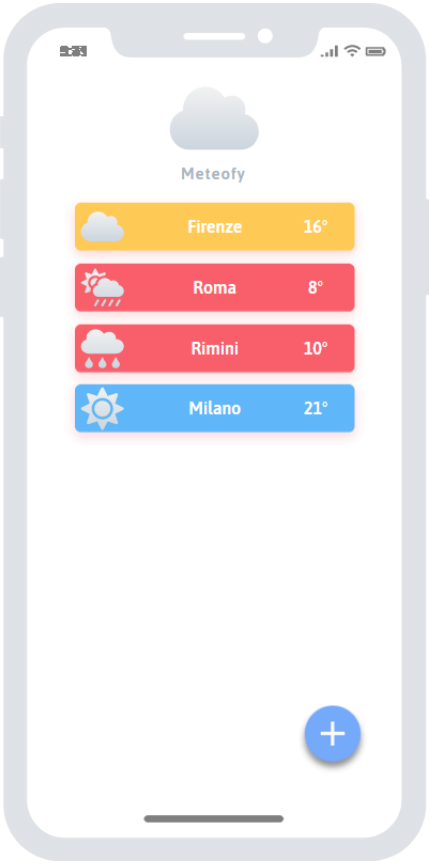

Figura 4.1: Progetto dell'interfaccia utente

4.3.3 Memorizzazione dei dati

I dati che vengono ricevuti dal servizio di OpenWeatherMap necessitano di essere salvati, per permettere di essere mostrati anche dopo la chiusura dell'applicazione. Risulta necessaria la presenza di un database interno all'applicazione, che sia persistente.

4.4 Implementazione

In questa sezione verrà descritto concretamente come sono state implementate le scelte progettuali citate nella precedente sezione.

Il codice sorgente della applicazione è disponibile su *GitHub*.

4.4.1 Architettura

Per lo sviluppo della applicazione è stato scelto come pattern architetturale il pattern *Model-View-ViewModel (MVVM)*, variante del pattern *Model-View-Controller (MVC)*, uno dei più noti pattern architetturali, il quale ha come obiettivo quello di disaccoppiare l'interfaccia utente dal modello dei dati, in modo da ottenere un'architettura più flessibile. Questo pattern, l'*MVC*, si basa sulla separazione logica degli aspetti dell'applicazione in tre componenti:

- **Model**: rappresenta il punto di accesso ai dati, che può essere una o più classi le quali leggono dati dal database oppure da un servizio Web.
- **View**: rappresenta la vista dell'applicazione, la sua interfaccia grafica, contenente tutti gli elementi che mostrano i dati, come *label*, caselle di testo e immagini.
- **Controller**: rappresenta il punto di incontro tra Model e View, in quanto i dati che questo riceve saranno elaborati per essere passati alla View.

A differenza dell'*MVC*, il *Model-View-ViewModel* assegna un ruolo più attivo alla *View*, in quanto questa è in grado di gestire gli eventi, eseguire delle operazioni ed effettuare il *data binding*. In questo contesto, quindi, alcune delle funzionalità del *Controller* vengono inglobate nella *View*, la quale si appoggia su un'estensione del *Model*: il *ViewModel*, da cui deriva il nome del pattern, il quale va a sostituire il *Controller*. Di seguito la classe `WeatherPlaceViewModel` usata nello sviluppo del progetto.

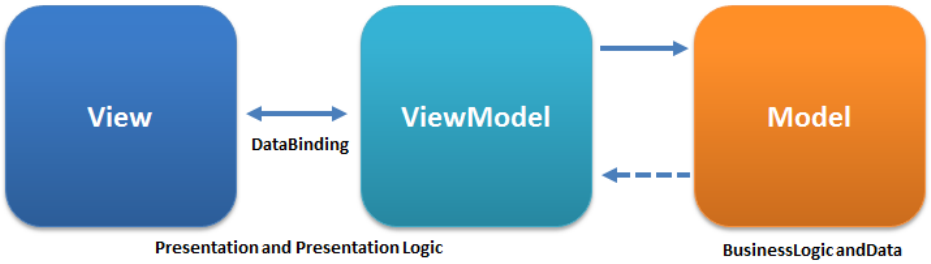

Figura 4.2: Pattern MVVM

Listing 4.1: Classe WeatherPlaceViewModel

```
class WeatherPlaceViewModel(private val repo: WeatherPlaceRepository):
    ViewModel() {

    val places: LiveData<List<WeatherPlace>> = repo.places

    //Permette l'inserimento di un nuovo WeatherPlace nel database
    //tramite Coroutine
    fun insertPlace(place: WeatherPlace) = viewModelScope.launch {
        repo.insertPlace(place)
    }

    //Permette di eliminare un elemento dal database tramite Coroutine
    fun deletePlace(placeId: String) = viewModelScope.launch {
        repo.deletePlace(placeId)
    }
}
```

4.4.2 Interfaccia Utente

L'intera applicazione è stata realizzata in una sola *Activity*, ovvero in una sola "pagina", in quanto non vi era la necessità di dividere l'interfaccia in più pagine. Nella sola Activity presente,la `MainActivity`, sono state inserite due componenti principali, oltre al logo e al nome della applicazione. La prima è la lista delle città presenti nel database interno all'applicazione, la

quale è stata implementata tramite `RecyclerView`. Una `RecyclerView` è uno widget presente su Android, che permette di mostare a schermo una lista dinamica di elementi. Gli elementi che vengono mostrati, in questo caso, sono composti da una immagine, contenente l'icona relativa alla condizione metereologica, il nome della città e la temperatura. Tali elementi verranno "riciclati", invece di venire distrutti, nel momento in cui non saranno più visibili a schermo, in un ottica di efficienza e riuso delle risorse grafiche. Il contenuto della lista rispecchia, in tempo reale, il contenuto del database interno. Ciò avviene grazie ad un *Observer* collegato alla lista delle città che viene aggiornata ad ogni inserimento.

Il secondo elemento importante nella schermata principale è il pulsante che permette all'utente di interagire con l'applicazione ed effettuare ricerche tramite il servizio di *OpenWeatherMap*. Tale pulsante è stato implementato come `FloatingActionButton`, che verrà indicato come *FAB*, ovvero un pulsante che rimane a schermo nonostante il movimento che subisce il contenuto sottostante. Collegato al *FAB* vi è un ascoltatore, necessario per gestire i click dell'utente su di esso. Ad ogni evento di click, viene creata e mostrata a schermo una istanza della classe `InputSheet`, dalla libreria *Sheets*. Una `InputSheet` permette di mostrare a schermo una campo per ricevere un input testuale dall'utente, il quale verrà validato ed passato ad un oggetto della classe `OpenWeatherMapCaller` il quale gestirà la comunicazione con le *API* di *OpenWeatherMap*.

Di seguito la classe `MainActivity`, usata per la gestione dell'interfaccia utente nello sviluppo dell'applicazione, e la classe `RecyclerAdapter`, usata per gestire il contenuto della `RecyclerView`.

Listing 4.2: Classe MainActivity

```
1 class MainActivity : AppCompatActivity() , CardClickListener{
2
```

```
3     private lateinit var mainAdapter: RecyclerAdapter
4     private lateinit var weatherPlaceViewModel : WeatherPlaceViewModel
5
6     override fun onCreate(savedInstanceState: Bundle?) {
7         super.onCreate(savedInstanceState)
8         setContentView(R.layout.activity_main)
9
10        initUI()
11        initViewModel()
12    }
13
14    private fun initUI(){
15        initRecyclerView()
16        initFAB()
17    }
18
19    //Inizializza la RecyclerView e il suo Adapter
20    private fun initRecyclerView() {
21        val mRecycler: RecyclerView = findViewById(R.id.main_recycler)
22        mainAdapter = RecyclerAdapter(this, ArrayList(), this)
23        mRecycler.layoutManager = LinearLayoutManager(this)
24        mRecycler.adapter = mainAdapter
25    }
26
27    //Inizializza il FAB
28    private fun initFAB(){
29        findViewById<FloatingActionButton>(R.id.fab).setOnClickListener {
30            run {
31                openSheets()
32            }
33        }
34    }
35
36    //Inizializza il ViewModel
37    private fun initViewModel(){
38        weatherPlaceViewModel = WeatherPlaceViewModel(WeatherPlaceRepository
39                .getRepositoryInstance(MeteofyDatabase
40                        .getDatabaseInstance(this)
41                        .weatherPlaceDAO()))
42        weatherPlaceViewModel
43                .places
```

```
44                .observe(this, {
45                    places -> mainAdapter.itemsHasChanged(places)
46                })
47    }
48
49    //Gestisce l'inserimento dei dati da parte dell'utente
50    private fun openSheets(){
51       InputSheet().show(this){
52            title("Aggiungi Citta'")
53            with(InputEditText {
54                required(true)
55                label("Inserisci nuova citta'")
56            })
57            onPositive { result -> result.getString("0")
58                    ?.let { OpenWeatherMapCaller(it,
59                            weatherPlaceViewModel, this@MainActivity) }
60            }
61        }
62    }
63
64    //Gestisce il tocco breve sulla singola cella,
65    //permettendo di aggiornare i dati sulle condizioni metereologiche
66    //della singola citta'
67    override fun onCardClickListener(data: WeatherPlace) {
68        OpenWeatherMapCaller(data.placeName,
69                weatherPlaceViewModel, this@MainActivity)
70    }
71
72    //Gestisce il tocco prolungato sulla singola cella,
73    //permettondo di cancellare tale cella e il relativo contenuto
74    //dal database locale
75    override fun onCardLongClickListener(data: WeatherPlace) {
76        weatherPlaceViewModel.deletePlace(data.placeName)
77    }
78}
```

Listing 4.3: Classe RecyclerAdapter

```
1  class RecyclerAdapter(private val context: Context,
2                        private var weatherPlaceList:
3                              MutableList<WeatherPlace>,
4                        private val cellClickListener: CardClickListener) :
```

```
5                               RecyclerView
6                                   .Adapter<RecyclerAdapter
7                                       .WeatherPlaceHolder>() {
8
9
10      override fun onCreateViewHolder(parent: ViewGroup,
11                                      viewType: Int): WeatherPlaceHolder{
12          return WeatherPlaceHolder(
13                  LayoutInflater.from(context)
14                          .inflate(R.layout.card_layout, parent, false))
15      }
16
17      override fun onBindViewHolder(holder: WeatherPlaceHolder,
18                                    position: Int) {
19          val currentPlace = weatherPlaceList[position]
20          populateViewHolder(holder, currentPlace)
21
22          holder.cardView.setOnClickListener {
23              cellClickListener.onCardClickListener(currentPlace)
24          }
25          holder.cardView.setOnLongClickListener {
26              cellClickListener.onCardLongClickListener(currentPlace)
27              true
28          }
29
30      }
31
32      private fun populateViewHolder(holder: WeatherPlaceHolder,
33                                     place : WeatherPlace){
34          holder.placeName.text = place.placeName
35          holder.placeTemp.text = place.placeTemp
36          holder.weatherImage
37                  .setImageDrawable(getWeatherImage(place.placeWeather))
38          holder.cardView
39                  .setCardBackgroundColor(getCardBackground(
40                          place.placeWeather)
41                  )
42      }
43
44      private fun getCardBackground(weatherType: String): Int{
45          return when(weatherType) {
```

```
46              "Rain" -> Color.parseColor("#ff5722")
47              "Clear" -> Color.parseColor("#0288d1")
48              "Snow" -> Color.parseColor("#ff5722")
49              "Clouds" -> Color.parseColor("#fbc02d")
50              "Fog" -> Color.parseColor("#fbc02d")
51              else -> Color.parseColor("#8bc34a")
52          }
53      }
54
55      @SuppressLint("UseCompatLoadingForDrawables")
56      private fun getWeatherImage(weatherType: String): Drawable?{
57          return when(weatherType){
58              "Rain" -> context.getDrawable(R.drawable.icon_rain)
59              "Clear" -> context.getDrawable(R.drawable.icon_sun)
60              "Snow" -> context.getDrawable(R.drawable.icon_snow)
61              "Clouds" -> context.getDrawable(R.drawable.icon_cloud)
62              "Fog" -> context.getDrawable(R.drawable.icon_cloud)
63              else -> context.getDrawable(R.drawable.icon_default)
64          }
65      }
66
67      fun itemsHasChanged(newPlaces: List<WeatherPlace>){
68          weatherPlaceList.clear()
69          weatherPlaceList.addAll(newPlaces)
70          notifyDataSetChanged()
71      }
72
73      override fun getItemCount(): Int {
74          return weatherPlaceList.size
75      }
76
77      class WeatherPlaceHolder(v: View) : RecyclerView.ViewHolder(v){
78          var placeName : TextView = v.findViewById(R.id.city_name)
79          var placeTemp : TextView = v.findViewById(R.id.city_temp)
80          var weatherImage : ImageView = v.findViewById(R.id.weather_type)
81          var cardView: CardView = v.findViewById(R.id.cardView)
82      }
83  }
```

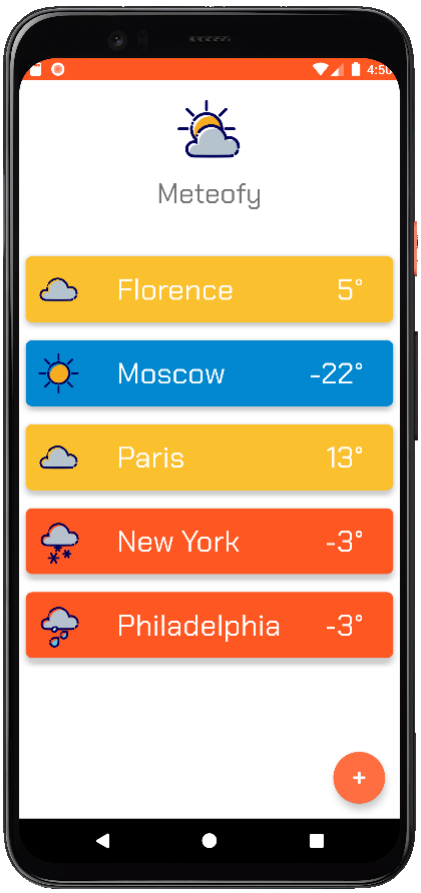

Figura 4.3: Implementazione finale dell'interfaccia utente

4.4.3 Recupero ed elaborazione dei dati

Come esposto in precedenza, i dati riguardanti le informazioni metereologiche delle varie città vengono ottenuti da *OpenWeatherMap*, tramite un servizio di *API*. Le richieste effettuate sono di tipo REST, le quali sfruttano il protocollo HTTP per il trasferimento delle informazioni. Per implementare la comunicazione Client-Server viene usata la libreria *OkHttp*, più precisamente la versione 3. Le richieste che vengono effettuate dall'applicazione sono di tipo GET e hanno una struttura del tipo:

Listing 4.4: Esempio di richiesta GET effettuata dall'applicazione

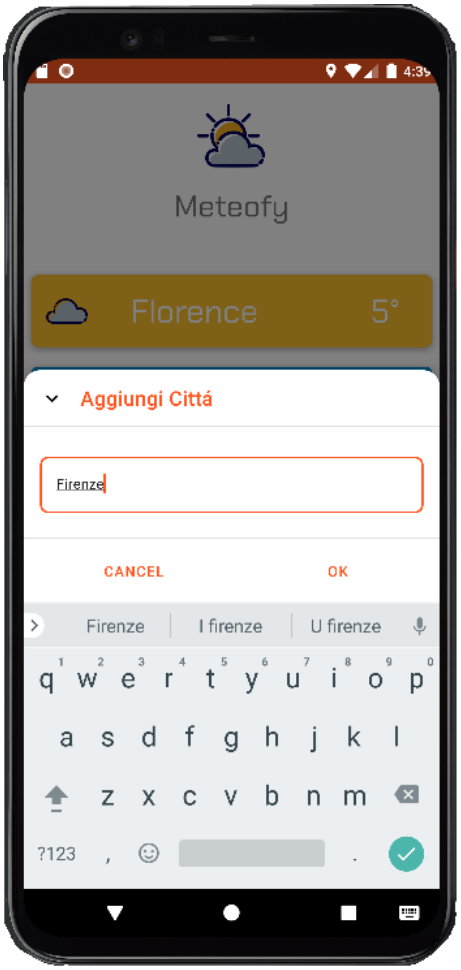

Figura 4.4: Inserimento dati da parte dell'utente

http://api.openweathermap.org/data/2.5/weather?q=Florence&appid=apiKey&units=metric

La risposta a questo tipo di richiesta contiene varie informazioni, in formato JSON, che necessitano di essere elaborate per estrapolarne quelle utili al fine dell'applicazione, come la temperatura e le condizioni metereologiche.

Per ottenere dei dati strutturati in modo da facilitarne l'elaborazione, è necessario convertire il codice JSON in classi Kotlin, effettuandone una deserializzazione. Per farlo viene usata la libreria di Google *GSON*. Da notare che GSON effettua la conversione da JSON a classi Java, le quali però, grazie

alla loro interscambiabilità, possono essere usate come classi Kotlin.

Di seguito il codice relativo alla classe dedicata al recupero dei dati e al loro salvataggio.

Listing 4.5: Classe OpenWeatherMapCaller

```
class OpenWeatherMapCaller(private val placeName: String,
                           private val weatherPlaceViewModel:
                                WeatherPlaceViewModel,
                           private val context: Activity)
                                : InternetCaller() {

    //OWM private API key
    private val apiKey : String = "myApiKey"

    init {
        startConnection()
    }

    override fun getPageUrl(): String {
        return "/data/2.5/"
    }

    override fun getHostingUrl(): String {
        return "http://api.openweathermap.org"
    }

    override fun getParameters(): String {
        return "weather?q=$placeName&appid=$apiKey&units=metric"
    }

    override fun onResponseReceived(response: String) {
        //OWM invia un messaggio contenente il codice '404'
        //nel caso in cui la richiesta non abbia risultati
        if(response.contains("404")){
            onErrorReceived("Citta non presente nel DB")
        }else {
            //viene effettuata la deserializzazione da JSON a oggetti Java
            //e poi questi oggetti vengono elaborati ed estratti
            //i dati necessari
            consumeRawPlace(
```

```
36                      Gson().fromJson(response, OWMJsonResponse::class.java)
37              )
38          }
39      }
40
41      override fun onErrorReceived(error: String) {
42          context.runOnUiThread {
43              Toast.makeText(context, error, Toast.LENGTH_SHORT).show()
44          }
45      }
46
47      private fun consumeRawPlace(rawPlace: OWMJsonResponse) {
48          val placeTemp = rawPlace.main.feels_like.roundToInt()
49          val placeName = rawPlace.name
50          val weatherType = rawPlace.weather[0].main
51          weatherPlaceViewModel.insertPlace(
52                  WeatherPlace(placeName, "$placeTemp", weatherType)
53          )
54      }
55  }
```

4.4.4 Memorizzazione dei dati

Una volta ricevuti i dati ed averli elaborati, questi vengono inseriti nel database interno. Ciò viene effettuato tramite l'uso di una istanza del *View-Model*, che dialoga con l'interfaccia del database, il *Data Access Object*.
Per la gestire la memorizzazione dei dati viene usata la libreria di Google *Room*, la quale permette di gestire il database SQLite in modo semplice, astraendo il database reale come un database ad oggetti. Questo tipo di procedura è detta ***Object-relational mapping***. *Room* infatti permette di accedere al database tramite un *Data Access Object*, ovvero una interfaccia contenente le funzioni che posso essere eseguite sul database, alle quali sono associate le relative query SQL.
Di seguito il *Data Access Object* che è stato utilizzato in questo progetto.

Listing 4.6: Data Access Object

```
@Dao
interface WeatherPlaceDAO {
    @Query("SELECT * FROM WeatherPlaces")
    fun getWeatherPlacesAsLiveData(): LiveData<List<WeatherPlace>>

    @Query("SELECT count(*) FROM WeatherPlaces")
    fun getPlacesCount(): Int

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertNewPlace(newPlace: WeatherPlace)

    @Query("DELETE FROM WeatherPlaces WHERE placeName = :currentPlaceName")
    suspend fun deleteByPlaceId(currentPlaceName: String)
}
```

4.4.5 Testing

Per effettuare dei test efficaci, come descritto nella sezione 3.3.4, risulta necessario effettuarli sia per l'interfaccia utente che per i metodi unitari che hanno possibilità di provocare errori. Per eseguire dei test sulla interfaccia utente è stata usata la libreria, messa a disposizione da Google proprio per questo scopo, *Espresso*. Questa libreria permette di descrivere le interazioni con l'interfaccia e i risultati aspettati da questa queste interazioni.

Listing 4.7: Esempio di test effettuato con Espresso

```
@Test
fun greeterSaysHello() {
    onView(withId(R.id.name_field)).perform(typeText("Steve"))
    onView(withId(R.id.greet_button)).perform(click())
    onView(withText("Hello Steve!")).check(matches(isDisplayed()))
}
```

Le interazioni possibili nell'applicazione si limitano a quelle con l'unico pulsante presente nella schermata principale, l'inserimento dei dati nella casella

di testo che appare in seguito e l'interazione con la liste delle città (apparizione ed eliminazione).

Per effettuare i test sui metodi unitari è stato usato *JUnit 4* e il framework *Robolectric* nei casi in cui fosse necessario l'utilizzo di parametri relativi all'applicazione simulata, come il contesto dell'applicazione.

Di seguito i test effettuati sul database e sulla classe `OpenWeatherMapCaller`.

Listing 4.8: Test relativo alla classe OpenWeatherMapCaller

```
class OpenWeatherMapCallerTest : InternetCaller(){

    private val testPlaceName: String = "Firenze"
    private val testPlaceNameWithError: String = "RandomPlaceThatNotExist"
    private val apiKey: String = "myApiKey"
    private val urlTest: String =
            "http://api.openweathermap.org/data/2.5/weather?q=" +
                    "$testPlaceName&appid=$apiKey&units=metric"

    //Test di una richiesta valida verso OWM
    @Test
    override fun startConnection() {
        val call = OkHttpClient().newCall(Request.Builder()
                .url(getHostingUrl() + getPageUrl() + getParameters())
                .build())
        Assert.assertEquals(call.request().url().toString(), urlTest)

        call.enqueue(object : Callback {
            override fun onFailure(call: Call, e: IOException) {
                onErrorReceived("Errore nel collegamento al DB")
            }

            override fun onResponse(call: Call, response: Response) {
                response.body()?.string()?.let {
                    Assert.assertNotNull(it)
                    onResponseReceived(it)
                }
            }
        })
    }

```

```
32      //Test di una richiesta non valida (contenente un errore) verso OWM
33      @Test
34      fun startConnectionWithError() {
35          OkHttpClient().newCall(Request.Builder()
36              .url(getHostingUrl() + getPageUrl() + getParametersWithError())
37              .build()).enqueue(object : Callback {
38              override fun onFailure(call: Call, e: IOException) {
39                  onErrorReceived("Errore nel collegamento al DB")
40              }
41
42              override fun onResponse(call: Call, response: Response) {
43                  Assert.assertNotNull(response)
44                  response.body()?.string()?.let {
45                      Assert.assertTrue(it.contains("404"))
46                      onResponseReceived(it)
47                  }
48              }
49          })
50      }
51
52      private fun getParametersWithError(): String {
53          return "weather?q=$testPlaceNameWithError" +
54                  "&appid=$apiKey&units=metric"
55      }
56
57      override fun getParameters(): String {
58          return "weather?q=$testPlaceName&appid=$apiKey&units=metric"
59      }
60
61      override fun getPageUrl(): String {
62          return "/data/2.5/"
63      }
64
65      override fun getHostingUrl(): String {
66          return "http://api.openweathermap.org"
67      }
68
69      override fun onResponseReceived(response: String) {
70          if (response.contains("404")) {
71              onErrorReceived("Citta non presente nel DB")
72          }else {
```

```
73              val rawPlace = Gson().fromJson(response,
74                      OWMJsonResponse::class.java)
75              Assert.assertNotNull(rawPlace)
76              Assert.assertEquals(rawPlace.name, "Florence")
77          }
78      }
79
80      override fun onErrorReceived(error: String) {
81          Assert.assertNotNull(error)
82      }
83  }
```

Listing 4.9: Test relativo alle funzioni del database

```
1   @RunWith(RobolectricTestRunner::class)
2   class MeteofyDatabaseTest {
3       private val testData = WeatherPlace("test_place",
4               "10", "Clouds")
5       private val appContext : Context = InstrumentationRegistry
6               .getInstrumentation().targetContext
7
8       private lateinit var db : MeteofyDatabase
9       private lateinit var dao: WeatherPlaceDAO
10      private lateinit var testObserver : TestObserver<List<WeatherPlace>>
11
12      @get:Rule
13      val testRule = InstantTaskExecutorRule()
14
15      @Before
16      fun setup() {
17          db = Room.inMemoryDatabaseBuilder(appContext,
18                  MeteofyDatabase::class.java)
19                  .allowMainThreadQueries()
20                  .build()
21          dao = db.weatherPlaceDAO()
22          testObserver = dao.getWeatherPlacesAsLiveData()
23                  .test()
24      }
25
26      @After
27      @Throws(IOException::class)
28      fun tearDown() {
```

```
29          db.close()
30      }
31
32      @Test
33      fun insertAndDeleteData(){
34          insertData(testData)
35          dao.deleteByPlaceId(testData.placeName)
36          Assert.assertTrue(testObserver.value().isEmpty())
37      }
38
39      @Test
40      fun insertAndRetrieveData() {
41          insertData(testData)
42          Assert.assertTrue(
43                  testObserver.value()[0].placeName == testData.placeName
44          )
45      }
46
47      private fun insertData(data: WeatherPlace){
48          dao.insertNewPlace(data)
49      }
50  }
```

Capitolo 5

Conclusioni

5.1 Esito dello sviluppo in Kotlin

Lo sviluppo dell'applicazione Meteofy tramite il linguaggio Kotlin è risultato molto semplice, in quanto le conoscenze acquisite nello sviluppo di applicazioni precedenti tramite Java si è rivelato un valido aiuto. è stato notato infatti come Kotlin sia una linguaggio che va a migliore Java in vari aspetti, senza stravolgere le sue fondamenta.
Dopo una prima fase di approccio al linguaggio, nel quale risultava talvolta complicato capire cosa facesse una certa istruzione, muoversi all'interno di Kotlin è diventato molto semplice e piacevole, anche grazie all'enorme supporto che la community fornisce a che si trova alle prime armi con il linguaggio, tramite la presenza di molti siti web dedicati e, soprattutto, grazie alla facile fruizione e comprensione della documentazione ufficiale.
Nonostante l'adozione di Kotlin da parte della community Android sia molto recente, al momento della stesura di questo documento, il linguaggio mostra già le sue potenzialità e la sua completa integrazione nell'ecosistema Android.
Durante lo sviluppo non sono stati riscontrati svantaggi nell'uso di Kotlin ri-

spetto a Java, mentre è possibile elencare vari vantaggi, quali la gestione dei valori *null*, la semplicità nell'uso delle coroutines e la compattezza del codice.

Ringrazio Pietro, Bernardo, Andrea, Mohamed e Denny per avermi accompagnato in questo viaggio.

Bibliografia

Sitografia

[1] Documentazione ufficiale Kotlin - *kotlinlang.com/docs*

[2] Testing in Android - *developer.android.com*

[3] Usare Room in Kotlin - *developer.android.com/codelabs*

[4] RecyclerView in Kotlin - *raywenderlich.com*

[5] Libreria OkHttp - *square.github.io/okhttp*

[6] Libreria GSON - *github.com/google/gson*

[7] Libreria Room - *developer.android.com/jetpack/androidx/releases/room*

[8] Architetture in Android - *developer.android.com/jetpack/app-arch*

[9] Repository Pattern - *medium.com/swlh/repository-pattern*

[10] Testing in Android - *developer.android.com/testing/fundamentals*

Bibliografia

[11] Neil Smyth - *Android Studio 4.0 Development Essentials - Kotlin Edition*

[12] Antonio Leiva - *Kotlin for Android Developers*