ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

Programmazione Reattiva in Kotlin su sistemi Android

Tesi in Programmazione ad Oggetti

Presentata da
Nicolas Barilari

Relatore
Prof. Mirko Viroli

Correlatore
Prof. Danilo Pianini

Seconda Sessione di Laurea
Anno Accademico 2017 - 2018

Parole chiave

Programmazione reattiva

Kotlin

RxKotlin

Android

Alla mia famiglia

A Cecilia

A Nik

Sommario

La presente tesi di Laurea ha lo scopo di descrivere approfonditamente il paradigma di *programmazione reattiva* applicato al linguaggio di programmazione *Kotlin*, focalizzando la trattazione verso lo sviluppo di applicazioni Android ad alto livello di reattività e con forte presenza di flussi asincroni di dati ed eventi.

La tesi si articola in tre capitoli. Nel Capitolo 1 vengono descritte dettagliatamente le caratteristiche del paradigma di programmazione reattiva, partendo dai concetti su cui si basa, per poi porre l'attenzione sui suoi principali contesti di utilizzo, sui vantaggi rispetto ad altri approcci e sulle problematiche generate dal suo impiego distribuito. In seguito viene chiarito ciò che si intende per *Functional Reactive Programming*, mostrando le relazioni che questo instaura con il paradigma da cui eredita e le astrazioni basilari che introduce all'interno del panorama reattivo. Infine si espande la trattazione considerando aspetti generali quali la tassonomia dei linguaggi reattivi, l'elenco delle maggiori aziende che utilizzano tecnologie basate sul paradigma di programmazione reattiva e la descrizione del manifesto reattivo, con annessa spiegazione dei principi fondanti in esso espressi.

Il Capitolo 2 è invece riservato alla descrizione del linguaggio di programmazione Kotlin, partendo dalle motivazioni del suo utilizzo e sviluppo e dalle caratteristiche di base, fino ad arrivare alla sintassi vera e propria e alla descrizione delle sue particolari funzionalità. Vengono quindi ricoperti gli argomenti di maggior interesse quali le variabili, i tipi di dato, l'inferenza di tipo, la *strict null safety*, i meccanismi introdotti per il flusso di controllo, per le classi e per gli oggetti, le funzioni, le lambda, le estensioni e le coroutines. Infine vengono elencate e spiegate prevemente le novità introdotte con l'attuale ultima versione 1.2 di Kotlin.

Il Capitolo 3 descrive la libreria RxKotlin, che rende pratiche le conoscenze acquisite nei precedenti capitoli, mostrando un'implementazione della programmazione reattiva nel linguaggio Kotlin. L'intero capitolo prende ad esempio la programmazione di applicazioni Android, dove RxKotlin mette in mostra le sue più importanti funzionalità. Vengono quindi descritte differenze ed affinità con RxJava (suo gemello), le fasi necessarie per il suo utilizzo su *Android Studio*, i concetti di base, le caratteristiche più avanzate quali l'approccio al multithreading (grazie all'uso affiancato della libreria *RxAndroid*), la backpressure e gli operatori per la manipolazione dei flussi asincroni osservabili. Infine, viene proposto un breve confronto con le coroutines e vengono descritte le tecniche per testare il codice.

Indice

Introduzione

Gli argomenti inseriti nella presente tesi di Laurea sono stati scelti in maniera da fornire utili strumenti per la programmazione asincrona in ambienti ricchi di potenziali sorgenti di eventi e che necessitino di risposte reattive agli stimoli. *Android* è quindi subito sembrato uno tra i sistemi più adatti a tali scopi, a causa della presenza di interfaccia utente e di innumerevoli fonti generatrici di informazioni d'interesse (tocchi sul display touchscreen, coordinate provenienti dal sensore gps, animazioni, richieste internet, ecc..). Per quanto riguarda invece il linguaggio e il paradigma di programmazione utilizzati, è stato scelto *Kotlin* per il suo stile conciso e moderno e per il ruolo che riveste nel panorama Android (è infatti il primo linguaggio supportato sul sistema operativo mobile di Google da ottobre 2017) e la *Reactive Programming* per il suo approccio reattivo e asincrono (difficilmente eguagliabile con altri tipi di paradigmi), che ben si sposa con le nuove esigenze introdotte dalle attuali applicazioni per smartphone e dagli odierni servizi web. Infine, allo scopo di "fondere" insieme i concetti precedentemente elencati, rendendo l'intera trattazione maggiormente coesa e autonoma nei contenuti, è stata scelta la libreria *RxKotlin* (seconda versione, attualmente la più recente): un'implementazione scritta in Kotlin del paradigma di programmazione reattiva, largamente utilizzata soprattutto per lo sviluppo di moderne applicazioni Android.

Il paradigma di programmazione imperativa è attualmente quello più utilizzato[1] e si pone alla base della gran parte dei linguaggi di programmazione odierni comunemente impiegati. Esso affonda le radici attorno al concetto di *istruzione*: più istruzioni in ordine specificano passo passo come un programma debba raggiungere gli obiettivi prefissatigli dal programmatore. All'interno di questo paradigma trova posto il (sotto-)paradigma di programmazione procedurale, il quale prevede l'utilizzo all'interno di un programma di una o più procedure, cioè porzioni delimitate di codice contenenti una serie di passaggi computazionali da eseguire. Detto ciò, la maggior parte dei linguaggi più diffusi, compresi i linguaggi della programmazione orientata agli oggetti (*OOP, Object-Oriented Programming*) tra i quali Java, C++, C# e Visual Basic è progettata principalmente per supportare proprio la programmazione (procedurale) imperativa. Questo tipo di approccio è però poco efficiente nel caso di sviluppo di applicazioni che prevedano la presenza di molteplici flussi di dati ed eventi generati in maniera asincrona e che necessi-

[1] Per maggiori informazioni visitare: `http://archive.is/syFnD`

tino di elaborazioni e risposte reattive da parte, talvolta, di differenti thread in esecuzione concorrente (un esempio può essere un'applicazione Android che deve reagire prontamente e contemporaneamente ad eventi provenienti dalla rete, dall'utente e dal sistema stesso). Oggigiorno le moderne applicazioni web e mobile devono gestire una miriade di eventi in tempo reale e dialogare continuamente con i server, l'interfaccia grafica, gli eventuali sensori e molti altri componenti al fine di offrire all'utente un'esperienza altamente interattiva; la programmazione reattiva è in questo senso un'ottima risposta, in quanto si propone di risolvere con semplicità ed efficienza le problematiche che si vengono spesso a generare in queste situazioni e alle quali la programmazione imperativa tradizionale fatica a trovare rimedio. Proprio a causa delle sue potenzialità, il paradigma di programmazione reattiva, viene spesso impiegato da numerose aziende in tutto il mondo tra le quali *Microsoft*, *Netflix*, *GitHub*, *SoundCloud*, *Trello* e molte altre.

Il linguaggio di programmazione Kotlin, è diventato il terzo linguaggio ufficialmente supportato da Google per Android dopo Java e C/C#. Kotlin presenta uno stile conciso, espressivo e progettato per gestire con sicurezza i tipi (*type-safe*) e i valori nulli (*null-safe*).Concentrando la presente tesi sulle applicazioni mobile in ambito Android, appare naturale introdurre la programmazione reattiva sfruttando Kotlin.

Per permettere l'utilizzo della reactive programming con Kotlin su sistemi Android sono state utilizzate le librerie *RxAndroid* ed *RxKotlin* in versione 2 (estensione scritta in Kotlin di RxJava 2)[2].

[2]Per maggiori informazioni visitare: `http://archive.is/5eRMs`

Capitolo 1

Programmazione reattiva

Questo capitolo si propone di descrivere il panorama della *programmazione reattiva* partendo dal principio, cioè dalle sue caratteristiche fondanti, per poi procedere a piccoli passi verso argomenti più di dettaglio cercando sempre, prima dell'introduzione di ognuno di essi (per quanto possibile), di fornire al lettore le nozioni di base per poter affrontare la lettura senza difficoltà.

1.1 Definizione

Premettendo che non è tuttora esistente una definizione ufficialmente riconosciuta di *programmazione reattiva* (*RP, Reactive Programming*), una di quelle che condensa al meglio la filosofia di questo paradigma è probabilmente la seguente:

> *la programmazione reattiva è un paradigma di programmazione che si avvale dei concetti del pattern Observer per gestire ed elaborare dati ed eventi tramite flussi asincroni e osservabili dall'esterno, in accordo con la nozione di propagazione del cambiamento.*

1.2 Caratteristiche

Dalla definizione si può evincere come la programmazione reattiva sia un paradigma fortemente correlato all'utilizzo di stream asincroni di dati/eventi e perciò particolarmente utile nella programmazione event-driven, efficace per lo sviluppo di applicazioni che necessitano di risposte reattive a stimoli provenienti in maniera asincrona da molteplici fonti, situazione riscontrabile tipicamente su piattaforme web e mobile. Questo approccio prevalentemente asincrono indebolisce i legami (*loose coupling*) tra i componenti dei programmi, producendo codice altamente modulare e dunque più facilmente modificabile, manutenibile ed estensibile.

È importante precisare che si parla di flussi di informazioni osservabili, i cui dati ed eventi emessi sono perciò catturabili dall'esterno e gestibili all'occorrenza da tutti coloro che necessitino di continui aggiornamenti riguardo uno specifico

evento d'interesse. All'interno di questo contesto trova dunque facilmente posto il Design Pattern Observer [11] della GoF[1], il quale definisce una o più dipendenze fra oggetti, così che quando uno di essi cambia stato, tutti i suoi sottoscrittori vengono notificati automaticamente e possono reagire di conseguenza.

Per quanto riguarda invece il concetto di *propagazione del cambiamento* (*Propagation of change*), si tratta di una nozione presente in letteratura e su cui si fonda il pensiero della programmazione reattiva; verrà esposto in maniera più dettagliata nella prossima sottosezione.

1.2.1 Propagazione del cambiamento

Nel paradigma di programmazione reattiva tutto può essere trasmesso tramite stream asincroni: dati, eventi, messaggi, chiamate, situazioni di successo e di errore. Si possono osservare questi flussi e reagire quando da essi vengono emessi valori. Questo ha un effetto interessante sulle applicazioni poiché tende a farle diventare intrinsecamente asincrone ed event-driven, reattive nel gestire i cambiamenti in atto e le loro propagazioni spesso prive di sincronia. All'interno del contesto delineatosi si afferma l'idea di *propagazione del cambiamento*, concetto che sta alla base del paradigma in esame.

Si prenda come esempio esplicativo la seguente istruzione in pseudo-codice:

```
a := b + c
```

Se si utilizza un linguaggio imperativo, l'assegnamento di cui sopra applica alla variabile `a` il risultato dovuto alla somma tra le variabili `b` e `c`, e questo avviene nel preciso istante nel quale l'espressione viene valutata. Ciò significa che un cambiamento successivo dei valori di `b` e/o di `c` non produce alcun effetto sulla variabile `a`, la quale mantiene sempre il valore originariamente calcolato (a meno, ovviamente, di una nuova valutazione dell'espressione stessa). In programmazione reattiva, invece, il concetto di *propagazione del cambiamento* rende il procedimento differente: il valore di `a` viene infatti sempre aggiornato automaticamente ogniqualvolta i valori delle variabili `b` e/o `c` cambiano, senza prevedere la necessità da parte del programma di rieseguire l'istruzione di assegnamento di `a`.

La logica della propagazione del cambiamento può essere per esempio implementata per mezzo di un grafo i cui nodi rappresentano i valori da tenere aggiornati e i cui archi rappresentano le relazioni di dipendenza che coinvolgono i nodi. Tramite questo tipo di architettura si viene quindi a creare una rete di dipendenze computazionali ed il cambiamento di un nodo (cioè di un valore) scatena una reazione che implica l'attraversamento degli eventuali archi ad esso collegati e il conseguente aggiornamento di tutti i nodi raggiunti.

Per rendere più chiari questi concetti, viene esposto di seguito un semplice esempio in pseudo-codice:

[1] *Gang of Four*, così vengono spesso chiamati Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, autori del libro *Design Patterns: Elements of Reusable Object-Oriented Software.*

```
number1 := 4
number2 := 6
sum := number1 + number2
average := sum / 2
```

Il codice mostra l'assegnamento dei valori 4 e 6 a due variabili (`number1` e `number2`), con successivo calcolo e salvataggio della somma e della media aritmetica (rispettivamente all'interno delle variabili `sum` ed `average`). In questo caso risulta estremamente semplice costruire il corrispondente grafo delle dipendenze per la propagazione dei cambiamenti, in figura 1.1:

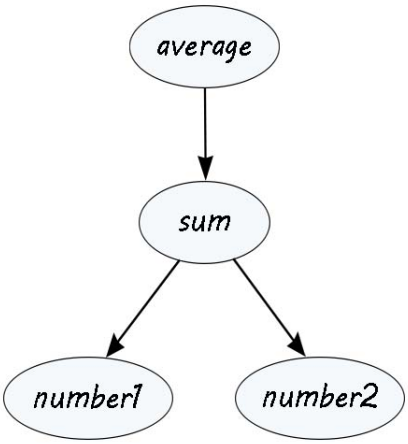


Figura 1.1: Grafo delle dipendenze, somma e media di due numeri

Si può facilmente notare come il primo nodo genitore (`average`) abbia solo un nodo figlio (`sum`), perciò dipenda solo da esso, mentre quest'ultimo dipenda invece da due nodi figlio (rappresentati dalle variabili `number1` e `number2`); questi ultimi sono attualmente anche nodi periferici del grafo orientato, perciò si possono considerare 'autonomi' perché non dipendono da nessun altro nodo figlio e nell'attuale configurazione non possono quindi mai subire modifiche propagate.

Seguendo l'approccio imperativo, i valori delle variabili `sum` ed `average` vengono assegnati durante l'esecuzione del frammento di codice e non risultano aggiornati dopo una modifica dei valori delle variabili che rappresentano i due numeri. Al contrario, seguendo l'approccio dettato dalla programmazione reattiva e in particolare dal concetto di propagazione del cambiamento su cui il paradigma si basa, ogni modifica dei valori delle variabili `number1` e `number2` propaga il cambiamento al nodo genitore `sum`, il quale aggiorna la somma e propaga a sua volta l'informazione di cambiamento al nodo genitore `average`, il quale infine aggiorna il valore della media aritmetica.

La situazione cambia leggermente nel prossimo esempio, sempre in pseudocodice:

```
numbers := [1, 2, 3, 4, 5, 6]
sum := 0
for each number in numbers
  sum := sum + number
end
average := sum / count(numbers)
```

In questo caso non sono presenti solo due numeri di cui far la somma e la media, bensì un array di numeri grande a piacere, perciò è necessario iterare l'intero vettore per calcolare la somma totale dei suoi elementi e ricavarne la lunghezza (`count(numbers)`) per quantificare il valore al denominatore nella formula per la media aritmetica. In figura 1.2 viene mostrato il grafo delle dipendenze per la propagazione dei cambiamenti relativo a questo esempio:

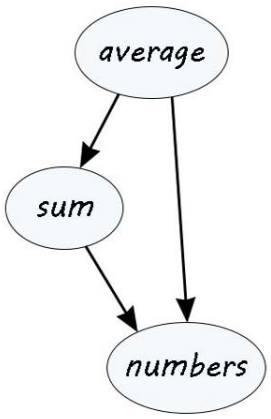


Figura 1.2: Grafo delle dipendenze, somma e media di un array di numeri

Si può osservare come il nodo genitore `average`, contrariamente all'esempio precedente, si ritrovi ora ad avere due nodi figlio da cui dipende, `sum` e `numbers`, e come quest'ultimo abbia due nodi genitore (`average` e `sum`). Questo accade per il semplice motivo che la variabile contenente la media viene assegnata tramite un'espressione che coinvolge entrambe le variabili (`average := sum / count(numbers)`); `average` si trova a dipendere quindi non più solo dal valore della somma ma anche dalla lunghezza dell'array di numeri. Se `numbers` subisce una mutazione, la notifica del cambiamento viene propagata attraverso il grafo delle dipendenze a `sum` ed `average`, ma solo il primo può essere aggiornato[2] poiché il secondo non possiede ancora tutte la nuove informazioni per poter calcolare la media; una volta che anche il nodo `sum` notifica il cambiamento del valore della somma al nodo genitore, allora questo viene aggiornato.

[2]Nel caso in cui `average` calcolasse comunque la media senza possedere tutte le informazioni aggiornate si potrebbe verificare un malfunzionamento dovuto ad inconsistenza dei dati chiamato *glitch*, argomento della sottosezione 1.2.4.

1.2.2 Strategie di propagazione

Le principali strategie adottabili per la propagazione del cambiamento nella programmazione reattiva sono le seguenti:

- **Complete propagation**
  Ogni nodo, durante la propagazione, invia il suo attuale stato *completo* ai nodi dipendenti; in questo caso, tutto lo stato precedente del nodo viene perso durante l'aggiornamento al nuovo stato. Questa strategia presenta un alto carico computazionale, in quanto ogni nodo mutato propaga il suo completo stato, perciò non è adatto nei casi in cui il grafo delle dipendenze computazionali sia particolarmente complesso o nel caso in cui gli stati necessitino di grandi quantità di dati per essere descritti.

- **Delta propagation**
  Nel momento della propagazione i nodi interessati inviano solo una porzione (un *delta*) del loro stato ai nodi dipendenti, cioè solo le frazioni di informazione che sono state effettivamente coinvolte da modifiche. Le porzioni di stato precedenti non soggette ad alterazioni permangono all'interno dei nodi, non vengono perse. Questo tipo di approccio tende a minimizzare la quantità di dati da trasportare lungo la catena di dipendenze durante la propagazione dei cambiamenti, dunque risulta efficiente anche nei casi in cui si presenti un grafo delle dipendenze complesso o una gran quantità di dati per la rappresentazione degli stati, situazioni nelle quali la *complete propagation* mostra invece limitazioni.

- **Batch propagation**
  La *batch propagation* è un approccio che prevede la non immediata propagazione dei cambiamenti, cioè la propagazione ritardata nel tempo. È un metodo di ottimizzazione: se due o più cambiamenti vicini nel tempo si annullano a vicenda (fanno cioè sì che si ritorni allo stato di partenza, precedente ai cambiamenti stessi), oppure un aggiornamento non modifica alcuno stato, la propagazione non avviene, in quanto non è in atto nessun cambiamento complessivo nel concreto. Al contrario, nel caso due o più cambiamenti vicini nel tempo generino un'effettiva mutazione di stato nel grafo, viene propagato solo l'ultimo cambiamento, infatti i precedenti risultano ormai obsoleti. Questa strategia minimizza il numero di propagazioni e quindi di percorrenza delle dipendenze nel grafo e di carico computazionale complessivo, ma tende a rendere il sistema meno reattivo ai cambiamenti, poiché essi vengono ritardati nel tempo. É dunque cruciale adottare un tempo di ritardo corretto, al fine di ritardare il cambiamento riducendo il carico di elaborazione ma al tempo stesso non diminuire eccessivamente la reattività del sistema; si delinea quindi una strategia efficace nel caso in cui le computazioni e il trasferimento di dati tra i nodi si rivelino particolarmente onerosi e sia tollerabile un leggero ritardo di reattività ai cambiamenti.

- **Invalidity notification propagation**

  Non si delinea propriamente come una strategia implementativa, quanto più come una "sotto-strategia", cioè un approccio utilizzabile potenzialmente accanto ad ognuna delle tre strategie precedentemente descritte. É utilizzabile inoltre soltanto nei casi in cui il sistema di interesse possieda un grafo delle dipendenze con direzionalità di propagazione di tipo *pull* oppure *hybrid* (verranno descritti nella prossima sottosezione). La *invalidity notification propagation* consiste nella richiesta di aggiornamento (*pull update*) da parte di un nodo che riscontra di possedere uno stato non valido, oppure che riceve un messaggio di propagazione non valido. In questo caso il nodo interessato scarta l'anomalia e richiede ai nodi da cui dipende un nuovo aggiornamento in merito. Dopo la ricezione del nuovo aggiornamento, e solo nel caso questo risulti valido, il nodo si aggiorna e lo propaga come di consueto attraverso le dipendenze ad esso collegate.

Si ricorda che è possibile anche progettare architetture reattive che contemplino la contemporanea presenza di più strategie, fra loro non in contrasto, al fine di ricavare vantaggi da tutte in base alle differenti situazioni. Un'ottima idea può essere per esempio quella di impiegare la *delta propagation* insieme alla *batch propagation*.

1.2.3 Modelli di valutazione

Per *modello di valutazione* (*evaluation model*) nella Reactive Programming si intende la dinamica direzionale adottata per la propagazione dei cambiamenti nel grafo delle dipendenze computazionali del sistema reattivo. Il modello viene applicato a livello di linguaggio e rimane trasparente al programmatore, che non deve quindi preoccuparsi di propagare i dati manualmente.

Come già specificato precedentemente, un nodo trasmette il cambiamento ad altri nodi suoi dipendenti seguendo la rete di relazioni rappresentata dal grafo; il nodo che invia può essere quindi visto come un produttore di dati (informazioni di cambiamento) e i nodi che ricevono come dei consumatori di dati, in accordo con la logica produttore-consumatore evidenziata in figura 1.3.

A questo punto, però, bisogna prendere un'importante decisione di progettazione: è necessario infatti scegliere quale modello di valutazione adottare, ognuno di essi offre vantaggi in alcune situazioni e svantaggi in altre. La letteratura presenta due principali modelli [1][9] a cui si aggiunge un terzo ibrido fra i primi due:

- **Push-based**

  Nel modello *push-based* la reazione ha inizio quando un nodo produttore (origine) cambia stato aggiornandosi, in questo caso "spinge" (push) l'informazione di cambiamento attraverso il grafo verso i nodi consumatori suoi dipendenti (destinazioni). É dunque un approccio guidato dalla disponibilità di nuove informazioni nei nodi origine (*data-driven*) e perciò presenta il

vantaggio di rendere il sistema altamente reattivo, in quanto le reazioni vengono provocate e propagate immediatamente dopo ogni disponibilità di nuovi dati. D'altra parte, però, questo approccio rivela il problema delle computazioni superflue e dell'alto carico di calcolo, poiché sono necessarie elaborazioni ogniqualvolta un nodo produttore cambia stato; inoltre introduce la possibilità di provocare *glitch* (si veda la sottosezione 1.2.4).

- **Pull-based**

  Nel modello *pull-based* sono i nodi consumatori che, quando necessitano di nuovi valori, li 'tirano' (pull) dai nodi sorgenti provocando una risposta da parte di questi ultimi. É dunque un approccio guidato dalla domanda di nuovi dati da parte dei nodi dipendenti (*demand-driven*) e perciò presenta il vantaggio di rendere il sistema maggiormente flessibile, in quanto i nodi che richiedono nuovi valori possono 'tirarli' all'occorrenza e non ad ogni cambiamento di questi. D'altro canto, però, l'approccio pull-based introduce latenze fra il momento nel quale si verifica un cambiamento e il momento nel quale avviene la corrispondente reazione di propagazione.

- **Hybrid push-pull**

  Il modello *hybrid push-pull* unisce i due modelli *push* e *pull* discussi precedentemente e tenta di ricavare benefici da entrambi gli approcci. Il modello *push-based* funziona bene in quelle parti di un sistema reattivo che necessitano reazioni istantanee al cambiamento, mentre il modello *pull-based* mostra migliori prestazioni nelle parti di sistema che presentano continui cambiamenti di valori nel tempo. L'approccio ibrido combinato guadagna i benefici del push-based (efficienza, reattività e bassa latenza) e del pull-based (flessibilità, basso carico computazionale e annullamento dei glitch).

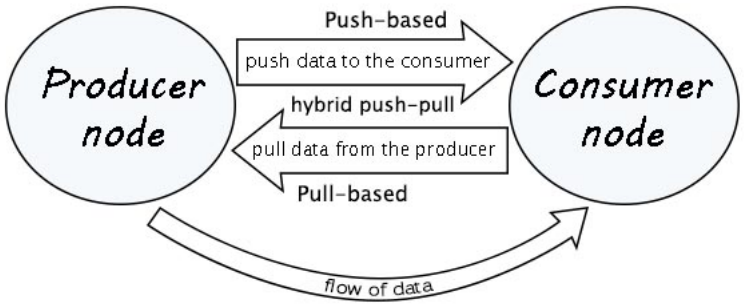


Figura 1.3: Possibili modelli di valutazione tra nodo produttore e nodo consumatore

1.2.4 Glitch

Un *glitch* in programmazione reattiva indica un malfunzionamento dovuto ad un'inconsistenza dei dati presenti nel grafo delle dipendenze computazionali. Può essere provocato da un'anomalia durante la propagazione di un cambiamento oppure, più di frequente, dal calcolo di un'espressione presente in un nodo prima della valutazione di tutte le sue dipendenze; viene quindi a crearsi una situazione per cui un nodo esegue la propria computazione utilizzando alcuni dati aggiornati ed altri ormai obsoleti, provocando quasi certamente errori di calcolo e inconsistenze. Di seguito un esempio per semplificare la comprensione, composto da pseudo-codice e relativo grafo delle dipendenze in figura 1.4:

```
var2 := var1 + 1
var3 := var1 + var2
```

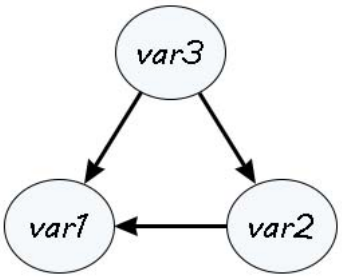


Figura 1.4: Grafo delle dipendenze che mostra un potenziale glitch

Nell'esempio, il valore (cioè lo stato) di `var3` dipende dai valori correnti delle variabili `var1` e `var2` mentre quest'ultima dipende solo da `var1`; infine `var1`, non possedendo archi uscenti, non dipende da nessuno, perciò è un nodo "autonomo".

In linea di principio, `var2` dovrebbe sempre possedere il valore di `var1` aumentato di un'unità mentre `var3` dovrebbe contenere sempre due volte il valore di `var1` più un'unità, qualunque sia lo stato di `var1` e anche a seguito di sue successive modifiche. Quindi, per fare qualche esempio pratico, a `var1 := 1` corrisponderebbero `var2 := 2` e `var3 := 3`, e nel momento in cui `var1` andasse incontro a una modifica, per esempio `var1 := 2`, `var2` e `var3` dovrebbero essere aggiornati di conseguenza ai valori, rispettivamente, di `3` e di `5`. Nella situazione in esame, però, la mutazione del valore contenuto in `var1`, con conseguente reazione di propagazione del cambiamento ai due nodi genitore, può creare un glitch. Questo si verifica se il nodo rappresentato da `var3` aggiorna il suo stato nel momento in cui riceve la notifica di cambiamento dal nodo `var1` ma quando ancora possiede il vecchio valore di `var2`; in questo caso specifico si viene a creare un'inconsistenza dei dati, in quanto `var1 := 2`, `var2 := 3` e `var3 := 4` invece che `5`.

É bene precisare che la situazione si normalizza quando `var3` riceve anche il nuovo valore dal nodo `var2`. Tuttavia il glitch rimane un problema da risolvere

perché in un dato istante, anche se breve, non vengono rispettate le relazioni di dipendenza esistenti nel sistema e si possono generare errori di calcolo a cascata in caso vengano utilizzati valori errati prelevati durante l'inconsistenza.

A questo punto dovrebbe essere semplice comprendere il motivo per cui i glitch possono manifestarsi solo nel caso in cui il linguaggio reattivo utilizzato impieghi un modello di valutazione di tipo push-based oppure ibrido. Accade infatti poiché in questi due modelli le sorgenti sono in grado di 'spingere' i cambiamenti verso i nodi dipendenti, ma non sono capaci di controllare il corretto aggiornamento proveniente dalle altre eventuali dipendenze (dei nodi genitore) a loro non collegate, rischiando quindi la propagazione di informazioni solamente parziali, che potrebbero portare alla nascita di glitch. Nel modello di valutazione pull-based, invece, ogni volta che un nodo necessita di nuovi dati li 'tira' ricevendoli da tutte le sue dipendenze, perciò non si presenta più il caso per cui viene propagata solo una parte delle informazioni e di conseguenza i glitch non possono manifestarsi.

Metodi di rimozione

Come già specificato, un semplice modo per evitare i glitch è quello di adottare un linguaggio reattivo che utilizzi il modello di valutazione pull-based.

Nel caso invece si voglia adottare un'altra tipologia di modello, occorrono alcuni accorgimenti: per eliminare del tutto il fenomeno del glitch, molti linguaggi di programmazione reattiva push-based e ibridi organizzano le espressioni tramite un grafo aciclico ordinato topologicamente; in questo modo viene fissato l'ordine d'esecuzione degli aggiornamenti nel grafo delle dipendenze ed è perciò assicurato che ogni espressione venga calcolata dopo che tutte le dipendenze ad essa collegate siano già state valutate.

Nella *teoria dei grafi* l'*ordinamento topologico* di un grafo consiste nell'ordinare linearmente i suoi nodi in modo che preso un qualunque arco xy dal nodo x al nodo y, x venga prima di y nell'ordinamento[3]. In altre parole, i nodi sono disposti in modo tale che ognuno di essi preceda tutti i nodi collegati ai suoi archi uscenti seguendo l'ordinamento scelto. Prerequisito perché un grafo possa essere ordinato topologicamente è che sia un *grafo aciclico orientato* o *grafo aciclico diretto* (*DAG, Directed Acyclic Graph*); in tal caso è garantito che esista almeno un suo ordinamento topologico, esempio in figura 1.5:

[3]Per maggiori informazioni visitare: `http://archive.is/vuDFT`

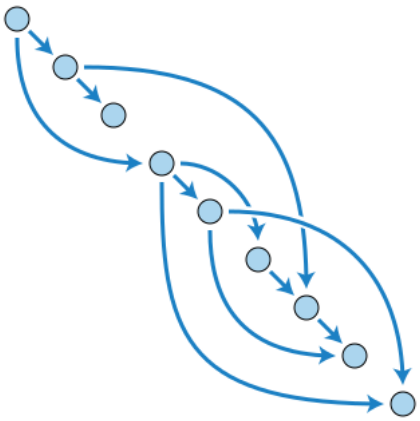


Figura 1.5: Grafo aciclico orientato (diretto) ordinato topologicamente partendo dal nodo in alto a sinistra verso quello in basso a destra

Ogni dipendenza presente in figura collega un nodo precedente ad uno successivo, non sono quindi mai presenti collegamenti 'all'indietro' (*back-edge*).

Si può dimostrare che se un grafo orientato può essere ordinato topologicamente (similmente alla figura di cui sopra), allora è senza dubbio aciclico, cioè non possiede una sequenza di archi diretti che permettano di partire da un qualsiasi nodo del grafo e tornare ad esso. In programmazione reattiva è fondamentale che il grafo delle dipendenze computazionali sia aciclico (e ovviamente orientato) poiché, in caso contrario, un evento di cambiamento verrebbe propagato all'infinito seguendo il ciclo formatosi, senza possibilità di conclusione.

Riprendendo l'esempio mostrato all'inizio di questa sezione ed illustrato in figura 1.4, è possibile ora rimuovere ogni glitch: il grafo è aciclico orientato (diretto) quindi è possibile ordinarlo topologicamente; per semplicità viene scelto come metodo di ordinamento quello che parte dall'alto a sinistra e si conclude in basso a destra, tuttavia anche alti metodi di ordinamento sono equivalentemente validi. Per agevolare la lettura viene di seguito rimostrato lo pseudo-codice e successivamente presentato il grafo ordinato, in figura 1.6:

```
var2 := var1 + 1
var3 := var1 + var2
```

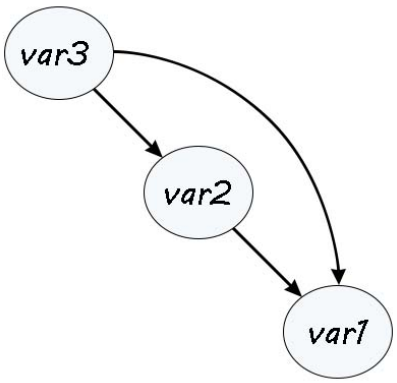


Figura 1.6: Grafo delle dipendenze ordinato topologicamente, esclude l'insorgenza di potenziali glitch

Come si può facilmente osservare, il grafo mostrato sopra mantiene la stessa struttura nodi-archi dell'esempio in figura 1.4, con la differenza che ora è ordinato topologicamente: preso ogni arco *xy*, il nodo `x` precede sempre il nodo `y`. Questo significa che adesso i nodi non possiedono più solo il proprio stato e le indicazioni sugli archi a loro collegati, ma anche un ordine di esecuzione ben preciso all'interno del grafo delle dipendenze; l'ordine garantirà la corretta esecuzione degli aggiornamenti dopo ogni propagazione di un cambiamento.

Ad ogni nodo viene perciò assegnata un'altezza (si tratta di un numero che può essere calcolato come l'altezza del nodo all'interno del grafo ordinato topologicamente) che è superiore a quella di qualsiasi nodo da cui dipende. I nodi vengono successivamente processati tramite una coda con priorità utilizzando proprio le altezze come fattore di priorità. Questa tecnica di rimozione dei glitch è comune a vari linguaggi di programmazione reattivi tra cui *Scala.React* e *Flapjax* [1].

Al nodo `var1` viene quindi applicata un'altezza `0`, a `var2` un'altezza `1` e a `var3` un'altezza `2` e vengono tutti posti in una coda seguendo la priorità data dal loro valore di altezza. Quando il nodo `var1` cambia stato la coda con priorità segna priorità pari a `0` e perciò permette al nodo di aggiornarsi ed inviare i dati ad entrambi i nodi genitore `var2` e `var3`. A questo punto la coda incrementa la priorità ad `1`, consentendo quindi soltanto a `var2` di aggiornare il proprio stato e propagare il cambiamento al suo unico nodo dipendente `var3`. Infine la coda incrementa la priorità a `2` permettendo a `var3` di aggiornarsi avendo ricevuto tutte le nuove informazioni e quindi eliminando la possibilità di generare glitch e inconsistenze di dati. La coda con priorità e l'ordine topologico imposto ai nodi del grafo si rivelano perciò cruciali, in quanto bloccano l'aggiornamento dei nodi che possiedono una priorità maggiore di quella attualmente servita dalla coda evitando che possano aggiornare prematuramente il proprio stato, prevedendo che essi possano avere in quel momento un quadro solamente parziale (e quindi sorgente di errori) degli aggiornamenti dei nodi dai quali dipendono.

1.2.5 Flussi asincroni di dati ed eventi

Il trasporto e l'elaborazione dei dati e degli eventi per mezzo di flussi asincroni osservabili dall'esterno è una caratteristica così importante da permeare l'intero paradigma di Reactive Programming. Nella programmazione reattiva, infatti, tutto può essere visto come una sorgente (un flusso) di dati: le variabili (che possono mutare stato emettendo un nuovo valore ad ogni cambiamento), le posizioni del puntatore del mouse, i tocchi dell'utente sul display dello smartphone, le coordinate del GPS, i messaggi provenienti dai social e molto altro. Qualsiasi struttura capace di incapsulare o inviare una qualsiasi tipologia di informazione può potenzialmente essere vista come uno stream di dati/eventi.

Per descrivere i flussi nel panorama della programmazione reattiva ci si avvale molto spesso dei cosiddetti *marble diagram*[4], diagrammi che semplificano la comprensione dell'andamento dei flussi di dati ed eventi nel tempo, del loro comportamento e delle operazioni applicate ad essi. Gli elementi costitutivi di base di un marble diagram sono i seguenti:

- una freccia/linea mono-direzionata che rappresenta lo scorrere del tempo (solitamente da sinistra a destra);
- delle piccole figure geometriche (spesso cerchi) giacenti sulla linea temporale, le quali rappresentano eventi oppure dati emessi dal flusso in un preciso istante di tempo e quindi ordinati temporalmente;
- una x posta sulla linea temporale che rappresenta l'invio di un messaggio di errore;
- un breve segmento ortogonale alla linea temporale e posto su di essa, il quale indica che lo stream ha terminato gli elementi da inviare e ha dunque concluso il suo compito.

Per fare un esempio, in figura 1.7 è rappresentato un flusso di eventi di click di un bottone attraverso il formalismo dei marble diagram:

[4]Per maggiori informazioni visitare: `http://rxmarbles.com/`

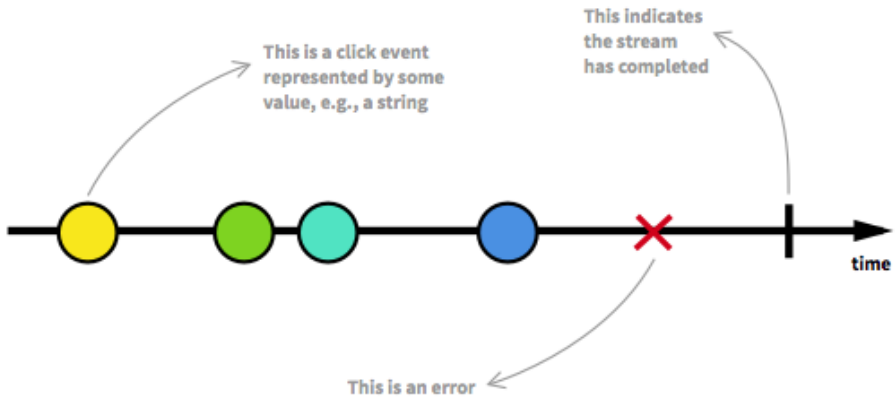


Figura 1.7: Esempio di marble diagram che rappresenta un flusso di eventi di click di un bottone

Dall'esempio si possono facilmente intuire le tre differenti tipologie di segnale (messaggio) che un flusso di dati è in grado di emettere:

- emissione di un elemento (dato oppure evento);
- emissione di un messaggio di errore;
- emissione di un messaggio di completamento.

Già ora, e in maggior misura nella prossima sotto-sezione, risulta chiaro come l'approccio utilizzato dalla programmazione reattiva non si basi esclusivamente su quello tipico del pattern Observer ma lo estenda includendo funzionalità nuove quali, un esempio, la capacità di notificare gli osservatori circa le condizioni di errore o completamento del flusso di dati.

1.2.6 Entità osservabili e osservatrici

I flussi (stream) di dati/eventi (per semplicità, d'ora in poi, il termine *dati/eventi* potrà essere sostituito talvolta con il termine *elementi*) nella programmazione reattiva vengono chiamati *entità osservabili*, mentre coloro che li osservano sono chiamati *entità osservatrici*. Un'*entità osservabile* rappresenta una sorgente di elementi mentre un'*entità osservatrice* è colei che rimane in ascolto di un'*entità osservabile*. É comprensibile che esistano queste due entità poiché, come già specificato, la Reactive Programming recupera i principi del pattern Observer.

Di seguito vengono descritte le due entità più nel dettaglio, mantenendo sempre il grado di astrazione tipico della letteratura sulla programmazione reattiva, dunque senza calare il discorso su un linguaggio o un'architettura specifici.

Entità osservabile

Un'*entità osservabile* rappresenta un flusso che emette dati/eventi all'esterno, ad intervalli regolari di tempo oppure in maniera totalmente asincrona; lo stream può avere un numero qualunque di elementi da emettere, inclusi zero o infiniti. Un'entità osservabile, però, non si limita solo ad emettere dati/eventi ma può anche terminare con successo inviando un *messaggio di avvenuto completamento*, oppure terminare con insuccesso inviando un *messaggio di errore*. Un'entità osservabile infinita (cioè con un numero potenzialmente infinito di elementi da emettere, come per esempio gli eventi di click dell'utente all'interno di un'applicazione Android) non potrà mai terminare con un messaggio di completamento, poiché questo deve essere inviato solo all'esaurimento degli elementi disponibili (in questo caso potenzialmente infiniti), potrà però terminare in ogni momento con un messaggio di errore nel caso in cui si manifesti un malfunzionamento durante l'esecuzione dell'entità osservabile stessa; in quest'ultima ipotesi l'entità infinita terminerà prematuramente e non emetterà più alcun elemento.

Entità osservatrice

Un'*entità osservatrice* si può legare (sottoscrivere) ad un'entità osservabile al fine di catturare i dati/eventi emessi da quest'ultima e gestirli secondo specifiche politiche. È bene ricordare che gli elementi vengono emessi *sempre* in sequenza, uno dopo l'altro, e l'entità osservatrice che li cattura ne gestisce solitamente uno solo dopo aver gestito il precedente. In generale quindi non si generano mai corse critiche[5]; se molti dati/eventi vengono emessi da un'entità osservabile in un breve intervallo temporale e l'entità osservatrice non riesce a stare al passo con la loro elaborazione, questi devono essere mantenuti in una coda degli eventi oppure scartati. Infine, un'entità osservabile può avere zero, una o più entità osservatrici sottoscritte.

1.3 Contesti d'utilizzo

I principali contesti nei quali risulta vantaggioso ed efficiente l'utilizzo della Reactive Programming sono i seguenti (e riassunti in figura 1.8):

- applicazioni interattive, dove è necessaria l'elaborazione di molteplici input provenienti dall'interfaccia grafica e azionati dall'utente in maniera asincrona (click del mouse o di un bottone, tocco sullo schermo dello smartphone, ecc..);
- applicazioni che interagiscono con sensori e perciò ricevono continui dati aggiornati in tempo reale (GPS, accelerometro, sensori di prossimità, ecc..);

[5]In realtà nulla impedisce al programmatore di sovrascrivere il normale comportamento di un'entità osservatrice, seppur possa essere fonte di problemi.

- applicazioni che instaurano connessioni multiple verso server e dunque necessitano di richiedere e inviare informazioni ad essi ininterrottamente;
- social network, che devono reagire prontamente alla rete e agli utenti, aggiornando in tempo reale i server e le applicazioni in base alle azioni degli utenti (like, post, upload di foto, chatting, ecc..) in maniera da riflettere immediatamente i cambiamenti su ogni dispositivo collegato;
- videogiochi, con multiple sorgenti di dati ed eventi da captare e gestire con elevate precisione e velocità;
- applicazioni per lo streaming video, che devono continuamente comunicare con i server e aggiornare lo stato ad ogni ricezione audio-video;
- applicazioni che recepiscono informazioni dalla rete e le visualizzano, per esempio applicazioni per il meteo, le news, RSS, ecc..;
- in generale tutte le applicazioni che ricevono informazioni da più sorgenti, specialmente se inviate in maniera asincrona, e che devono reagire prontamente ad ogni possibile evento di interesse.

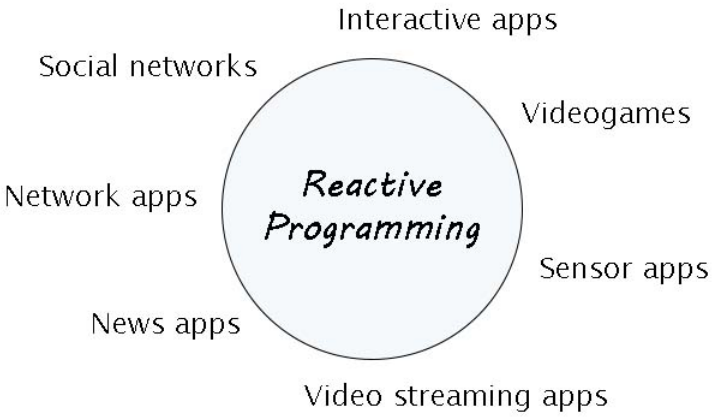


Figura 1.8: Principali contesti d'utilizzo della programmazione reattiva

1.4 Vantaggi rispetto ad altri approcci

Il paradigma di programmazione reattiva mostra i suoi grandi vantaggi specialmente se posto a confronto con quello di programmazione imperativa, in quanto essi risultano impiegare approcci diametralmente opposti in molteplici situazioni.

Nella programmazione imperativa è il programmatore a dover specificare, attraverso l'utilizzo di istruzioni, *come* il programma debba arrivare passo a passo al raggiungimento degli obiettivi prefissati; invece, utilizzando la programmazione

reattiva, ed in maniera ancora più evidente una sua estensione chiamata *programmazione reattiva funzionale*[6], l'approccio tende a farsi dichiarativo: in questo caso al programmatore viene richiesto di specificare principalmente *cosa* il programma deve fare, lasciando i dettagli del *come* farlo al linguaggio sottostante. La programmazione reattiva rende il codice più conciso e aumenta il livello di astrazione in modo che ci si possa concentrare sulla logica d'interdipendenza fra gli eventi, piuttosto che sul trattamento di grandi quantità di dettagli implementativi.

I vantaggi si rivelano ancora più evidenti se si osservano le moderne app web e mobile, altamente interattive e con una moltitudine di potenziali eventi provenienti dall'interfaccia grafica[7]. Una dozzina di anni fa, infatti, l'interazione con le pagine web consisteva essenzialmente nell'invio di lunghe form a backend e nell'esecuzione di semplici rendering a frontend[7]. Oggigiorno, invece, le app si sono evolute per essere real-time: la modifica di un singolo campo di una form può attivare automaticamente un salvataggio a backend, i 'like' ad alcuni contenuti sui social network possono essere riflessi in tempo reale ad altri utenti connessi e così via. Si sente quindi sempre maggiormente il bisogno di strumenti per affrontare queste situazioni in modo appropriato e la programmazione reattiva è in questo senso una possibile risposta. I linguaggi di programmazione imperativi (procedurali) in questo caso mostrano diversi segnali di debolezza, in quanto solitamente costringono a gestire la parte reattiva tramite pattern e questo approccio, a lungo andare, può rendere il codice più caotico, confuso e complicato da comprendere. La programmazione reattiva, invece, fornisce0 supporto diretto a livello di libreria/linguaggio, rendendo il codice più pulito, chiaro, robusto e manutenibile.

1.5 Programmazione reattiva distribuita

Con 'linguaggio di programmazione reattivo distribuito' si intende un linguaggio reattivo che permetta la ripartizione dei dati e del grafo delle dipendenze computazionali fra più nodi di una rete[8], in ambiente distribuito. Per esempio, nel caso di un'espressione del tipo `var3 := var1 + var2`, le tre variabili e il conseguente grafo delle dipendenze potrebbero trovarsi su nodi di rete differenti.

Sebbene la richiesta di linguaggi che supportino la programmazione distribuita sia alta (basti pensare alle applicazioni web e mobile che stanno diventando progressivamente sempre più distribuite), la progettazione di un linguaggio reattivo distribuito è considerevolmente più complicata rispetto a quella di un linguaggio che non supporti il distribuito. La maggior difficoltà risiede nel fatto che in questi contesti vanno presi in considerazione fattori quali la latenza, la congestione, i

[6] La *programmazione reattiva funzionale* (*FRP, Functional Reactive Programming*), sarà oggetto di discussione nella sezione 1.6.

[7] Per maggiori informazioni visitare: `http://archive.fo/iJVZh`

[8] Attenzione: con il temine *nodi*, in questo caso specifico, non ci si riferisce ai nodi del grafo delle dipendenze esistenti in Reactive Programming, bensì ai nodi di una rete presenti in un usuale sistema distribuito.

guasti della rete e la mancanza di un orologio globale, perciò gli attuali linguaggi di programmazione reattivi distribuiti spesso non possono garantire la completa rimozione delle inconsistenze dei dati nel grafo delle dipendenze (glitch). Tra i pochi linguaggi reattivi che supportano il distribuito troviamo *Flapjax*, *Lamport Cells* e *AmbientTalk/R* [1], tuttavia nessuno di essi assicura una completa rimozione dei glitch in caso di utilizzo in ambiente distribuito.

Per far fronte a guasti e ritardi della rete e aumentare la scalabilità, molti sistemi distribuiti sono implementati oggigiorno con approcci di tipo event-driven (per esempio le applicazioni che utilizzano la tecnologia *Ajax*, comunicando in maniera asincrona attraverso richieste di servizi web). Uno stile di comunicazione asincrono disaccoppia infatti le parti comunicanti: non devono essere connesse contemporaneamente per consentire la comunicazione e questo implica un aumento della robustezza dell'intero sistema. Tuttavia la necessità di un grafo delle dipendenze in un programma reattivo distribuito crea complicazioni (oltre alla presenza di glitch): tende infatti ad accoppiare strettamente i componenti dell'applicazione e quindi a renderli meno resistenti agli errori di rete e a ridurre la scalabilità complessiva.

Una possibile soluzione può essere quella di utilizzare un approccio centralizzato, predisponendo un'entità con un orologio centrale collegata a tutte le parti coinvolte del sistema reattivo distribuito e responsabile per l'ordine degli aggiornamenti nel grafo delle dipendenze. Questo metodo di accentramento introduce però un singolo punto di rottura (cosiddetto *single point of failure*) e probabilmente un sovraccarico di comunicazione considerevole poiché tutte le parti coinvolte nel sistema devono comunicare con un singolo host ogniqualvolta ricevono o propagano dati, limitando potenzialmente la scalabilità dell'intera architettura distribuita.

1.6 Programmazione reattiva funzionale

Sebbene sia spesso equiparato alla Reactive Programming, la *programmazione reattiva funzionale* (*FRP, Functional Reactive Programming*) è di fatto un paradigma autonomo, differente dal precedente, benché da esso erediti: si può infatti considerare come un'estensione "funzionale" della programmazione reattiva [25]. Molti linguaggi di programmazione reattivi offrono la possibilità di adottare approcci funzionali alla programmazione, perciò in questi casi si può ritenere più corretto parlare di programmazione reattiva *funzionale* anche se, come già accennato, non è raro vedere i due termini parificati.

É attualmente ritenuto accettabile utilizzare il termine Reactive Programming per riferirsi sia alla RP che alla FRP (guardare figura 1.9), in quanto di fatto la seconda è una specializzazione della prima; è invece concettualmente errato riferirsi ad entrambe con il nome di Functional Reactive Programming, in quanto così facendo si indica solo FRP.

Procedendo con ordine, risulta ora necessario esporre brevemente cos'è e cosa si intende per *programmazione funzionale* (*FP*, Functional Programming), al fine

di poter poi introdurre con maggior facilità la programmazione reattiva funzionale.

Functional Programming

Functional Programming è un paradigma di programmazione differente da quello imperativo (-procedurale) e che si contraddistingue principalmente per il suo approccio dichiarativo, per la fluidità e concisione e per la mancanza di side-effect nel codice. Le applicazioni implementate seguendo questo paradigma eseguono le computazioni attraverso una serie di funzioni matematiche trasparenti referenzialmente, cioè funzioni garanti che il loro risultato in output sia sempre identico per uno stesso insieme di parametri di input, indifferentemente dal momento e dalla posizione nelle quali vengono valutate; in questa maniera si eliminano gli effetti collaterali nel codice, comunemente presenti con l'utilizzo di linguaggi imperativi.

La programmazione funzionale, inoltre, non contempla cambiamenti di stato del programma, ma procede solo per trasformazioni di dati e applicazioni matematiche sugli stessi. Importante anche l'approccio che questo tipo di paradigma consente di attuare con gli stream, offrendo molteplici funzioni in grado di compiere mutazioni degli elementi in essi contenuti e di combinare in vario modo più stream di dati fra loro. Essi sono inoltre immutabili: ogni operazione su di essi ne produce uno nuovo.

Escludendo i linguaggi progettati per supportare un paradigma di programmazione specifico, gli altri linguaggi, cosiddetti 'generici', sono in genere sufficientemente flessibili da supportare più paradigmi, quindi talvolta anche la programmazione imperativa e funzionale all'interno dello stesso linguaggio, come ad esempio in Scala e in Python.

Functional Reactive Programming

Functional Reactive Programming [23] è un paradigma di programmazione che vede la sua originaria formulazione all'interno di un articolo pubblicato da Conal Elliott e Paul Hudak in occasione dell'*ICFP* (*International Conference on Functional Programming*) del 1997 ed intitolato *Functional Reactive Animation* [10]. Esso puntava alla progettazione e allo sviluppo di animazioni e grafica interattiva tramite il linguaggio di programmazione funzionale puro chiamato *Haskell* (nato nel 1990) e valse loro la premiazione come "articolo più influente dell'ICFP 97".

Successivamente alla sua nascita ha subito leggere modifiche e fisiologiche evoluzioni, fino ad arrivare a ciò che è tuttora: un paradigma che, come ben mostrato in figura 1.9, rende proprie caratteristiche tipiche della programmazione reattiva e di quella funzionale.

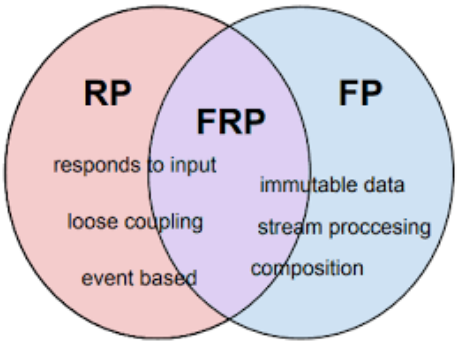


Figura 1.9: *Functional Reactive Programming* (al centro) in relazione a *Reactive Programming* (a sinistra) e *Functional Programming* (a destra).

La programmazione reattiva funzionale si configura quindi come un paradigma che fonda i suoi principi su quelli della programmazione reattiva ma imponendole un approccio totalmente funzionale, assimilando perciò anche tutte le caratteristiche (descritte precedentemente) che questo tipo di approccio implica.

É un paradigma che offre un elegante e conciso metodo per la creazione di programmi altamente interattivi e si propone di eliminare il cosiddetto fenomeno dello *spaghetti code*, comune nei tradizionali approcci di programmazione.

1.6.1 Astrazioni di base

La programmazione reattiva funzionale introduce due astrazioni importanti che consentono di modellare esplicitamente il tempo e i dati emessi in funzione di esso:

- **Behaviors (tempo-continui)**
  Chiamati anche *signals*, rappresentano flussi di valori che mutano e vengono generati continuativamente nel tempo, per questo la primitiva più semplice rappresentabile da un *behavior* può essere considerata proprio il tempo stesso. Questi flussi emettono elementi finché essi non terminano oppure si verifica un errore, dopodiché non emettono più altri dati. Nei linguaggi di programmazione reattivi funzionali i *behaviors* possono essere sempre espressi in funzione temporale. Esempi di entità generabili nel continuo e quindi rappresentabili tramite *behaviors* sono:
  - il tempo;
  - la posizione di un'immagine durante un'animazione;
  - la posizione del cursore del mouse;
  - la temperatura;
  - un audio;
  - un cronometro;

  - ...

- **Events (tempo-discreti)**
  Chiamati anche *event streams*, rappresentano flussi di valori che vengono generati in maniera discreta nel tempo, in momenti precisi. Anche questo genere di flussi emette elementi finché essi non terminano oppure si verifica un errore, ma ciò avviene in maniera temporalmente discreta. Esempi di entità generabili nel discreto e quindi rappresentabili tramite *events* sono:

  - il click del mouse;
  - la pressione di un bottone;
  - il tocco sul display touchscreen di uno smartphone;
  - il cambiamento di direzione del cursore del mouse;
  - l'apertura di una pagina web;
  - ...

I *behaviors*, a differenza degli *events*, sono quindi intrinsecamente scalabili in maniera arbitraria nel tempo nella stessa misura, per fare un esempio, nella quale le immagini vettoriali (indipendenti dalla risoluzione) lo sono nello spazio, contrariamente alle immagini raster (spazialmente discrete).

1.7 Tassonomia dei linguaggi reattivi

In letteratura i linguaggi di programmazione reattivi sono divisi in tre principali categorie [1]:

- **The FRP Siblings**
  I linguaggi di programmazione reattivi presenti in questa categoria sono quelli che utilizzano l'approccio funzionale dichiarativo, perciò sono identificabili esattamente come 'reattivi funzionali' e possiedono le caratteristiche e le funzionalità descritte nella sezione 1.6 (quindi le astrazioni di base della FRP, gli operatori per comporre e combinare flussi di dati/eventi fra loro, ecc..). La moderna ricerca inerente la programmazione reattiva si è focalizzata principalmente su questa categoria di linguaggi (basati sulla FRP), ed è soprattutto dovuto a questo il motivo per cui la maggior parte dei linguaggi reattivi attualmente esistenti sono funzionali.

Language	**Host language**
Fran [10]	Haskell
Yampa [13]	Haskell
Frappé [6]	Java
FrTime [5]	Racket
NewFran [9]	Haskell
Flapjax [21]	Javascript
Scala.React [19]	Scala
AmbientTalk/R [4]	AmbientTalk

Tabella 1.1: Alcuni dei principali linguaggi di programmazione che rientrano nella categoria *FRP Siblings*.

Nella tabella 1.1 (pagina precedente) sono elencati alcuni esempi [1] dei principali linguaggi di programmazione reattiva funzionale che rientrano nella categoria *FRP Siblings*.

- **The Cousins of Reactive Programming**

  I linguaggi di programmazione reattiva che rientrano in questa categoria sono ben pochi. Questi non supportano le astrazioni di base della Functional Reactive Programming per modellare valori espressi nel tempo, quali *behaviors* ed *events*, e non prevedono operatori per manipolarle e combinarle fra loro (come *map*, *switch*, *merge*, ...). Tuttavia possiedono il supporto per la propagazione automatica dei cambiamenti di stato e altre caratteristiche tipiche della Reactive Programming, come per esempio l'abilità di evitare i glitch. Nella seguente tabella 1.2 sono elencati alcuni esempi [1] dei principali linguaggi di programmazione reattiva che rientrano nella categoria *Cousins of Reactive Programming*:

Language	**Host language**
SuperGlue [20]	Java
Lamport Cells	E
Cells	Lisp
Coherence [8]	Coherence
Trellis	Python
.NET Rx [17]	C#

Tabella 1.2: Alcuni dei principali linguaggi di programmazione che rientrano nella categoria *Cousins of Reactive Programming*.

- **Synchronous, Dataflow and Synchronous Dataflow Languages**
  In quest'ultima categoria risiedono i linguaggi che vengono utilizzati per modellare sistemi reattivi in generale, perciò i linguaggi di programmazione che si basano sul paradigma di programmazione reattiva sincrona (*SRP*, *Synchronous Reactive Programming*) e su quello di programmazione del flusso di dati (*Dataflow programming*). Il primo paradigma si basa sull'idea che le reazioni siano istantanee, cioè non impieghino tempo, e siano atomiche; quest'ipotesi semplifica non poco i programmi, i quali possono quindi essere descritti con automi a stati finiti e tradotti in programmi modellati con linguaggi di tipo sequenziale. Il secondo paradigma citato, invece, esprime i programmi attraverso grafi orientati dove i nodi rappresentano le operazioni e gli archi rappresentano le dipendenze tra i dati e tra le computazioni. Nella seguente tabella 1.3 sono elencati alcuni esempi [1] dei principali linguaggi di programmazione che rientrano nella categoria *Synchronous, Dataflow and Synchronous Dataflow Languages*:

Language	**Host language**
Esterel [2]	Esterel
FairThreads [3]	C
StateCharts [12]	StateCharts
LabVIEW [14]	G

Tabella 1.3: Alcuni dei principali linguaggi di programmazione che rientrano nella categoria *Synchronous, Dataflow and Synchronous Dataflow Languages*.

1.8 Uso industriale

Non sono poche le aziende che hanno deciso di utilizzare librerie o linguaggi di programmazione basati sul paradigma di programmazione reattiva (o reattiva funzionale).

In figura 1.10 vengono mostrare alcune delle maggiori compagnie utilizzatrici di queste tecnologie, tuttavia quello mostrato è solo un piccolo sottoinsieme di tutte le aziende coinvolte, si tratta infatti soltanto di un campione di riferimento.

Figura 1.10: Alcune famose aziende che utilizzano tecnologie basate sulla programmazione reattiva

1.9 Il manifesto reattivo

Inizialmente ideato nel 2013, il 16 settembre 2014 è stata pubblicata la seconda versione del *The Reactive Manifesto*[9], un breve testo scritto da *Jonas Bonér*, *Dave Farley*, *Roland Kuhn* e *Martin Thompson*, che raccoglie le idee e le principali caratteristiche che un sistema reattivo dovrebbe possedere.

Da precisare che non si tratta del manifesto del paradigma di programmazione reattiva, ma soltanto di una guida alla quale attenersi nel caso si voglia progettare un robusto, manutenibile ed estendibile sistema reattivo. Non ha perciò particolare rilevanza se vengano utilizzati i principi della Reactive Programming oppure, per esempio, quelli alla base dei *sistemi ad attori* [15]: sono infatti molteplici gli approcci e le architetture potenzialmente adatti per la costruzione di sistemi reattivi. È però auspicabile, sebbene non automatico, che un programma implementato con tecnologie derivate dalla programmazione reattiva sia dotato di tutte le caratteristiche descritte nel presente manifesto.

I motivi che hanno portato alla sua stesura, a detta degli autori, sono i drastici cambiamenti degli ultimi anni: fino a pochi anni fa le applicazioni di grandi dimen-

[9]Per maggiori informazioni visitare: `http://archive.fo/8j2HL`

sioni erano infatti ancora caratterizzate dall'impiego di decine di server, dall'avere tempi di risposta nell'ordine dei secondi, dal richiedere ore di manutenzione offline e dal gestire pochi gigabyte di dati. Oggigiorno, invece, una miriade di applicazioni di ogni tipo girano su quasi ogni genere di dispositivo, da quelli mobili ai cluster basati sul cloud; l'unità di misura dei dati è oggi il petabyte e gli utenti di queste applicazioni si aspettano tempi di risposta nell'ordine dei millisecondi e un'autonomia di funzionamento pari al 100%. Perciò è semplice comprendere come le esigenze dei programmi di oggi non possano essere soddisfatte dalle architetture software del passato.

Di seguito sono descritte (e mostrate in figura 1.11) le quattro principali caratteristiche che un ottimo sistema reattivo deve possedere, secondo il manifesto:

- **Responsive**

  Un sistema reattivo deve essere efficiente, cioè deve rispondere agli eventi in maniera tempestiva, entro un tempo ragionevole al fine di aumentare il grado di usabilità e di utilità dell'intero sistema. La responsività si attua focalizzandosi sul minimizzare i tempi di risposta, individuando per ciascuno di essi un limite massimo prestabilito (scelto in base al contesto d'utilizzo) in modo da garantire una qualità del servizio costante nel tempo ed individuando e risolvendo prontamente gli eventuali problemi verificatisi durante l'esecuzione del servizio stesso. Il comportamento risultante diventa quindi predicibile, il che semplifica la gestione delle situazioni di errore, genera fiducia negli utenti finali e li predispone ad ulteriori utilizzi del sistema.

- **Resilient**

  Accade spesso che sistemi di cui sopra non siano però robusti in caso di guasto. Un sistema robusto risolve il problema premunendosi di componenti debolmente legati fra loro, dunque relegando i possibili malfunzionamenti all'interno di ogni componente, in modo tale da isolarli e da garantire che un guasto alle singole porzioni del sistema non comprometta l'intero sistema. Il recupero di ogni componente viene delegato ad un altro componente (esterno) e la disponibilità di continuo funzionamento viene assicurata tramite replica laddove necessario. Riassumendo, la resilienza si acquisisce quindi tramite isolamento, contenimento, delega e replica.

- **Elastic**

  L'elasticità di un sistema reattivo denota la sua capacità di "espandersi" e "contrarsi" adattandosi alle variazioni del carico di lavoro nel tempo, incrementando o decrementando le risorse allocate in base alle esigenze. Ciò porta ad architetture che non hanno né sezioni deboli né cosiddetti *colli di bottiglia*, favorendo così l'equa ripartizione degli input tra tutti i componenti. Questo tipo di approccio aumenta la scalabilità dell'intero sistema ed evita che si possano presentare squilibri di utilizzo tra le sue parti oppure, ancora peggio, che esse possano superare il limite massimo di carico di lavoro che sono in grado di gestire.

- **Message Driven**

  Uno dei modi per garantire un accoppiamento debole tra i componenti di un sistema reattivo, e quindi godere di tutti i vantaggi che questo comporta (isolamento, indipendenza, trasparenza, delega dei guasti, ...), è quello di basare l'architettura del sistema sullo scambio asincrono di messaggi. Questo approccio permette anche di semplificare la gestione del carico di lavoro, consentendo infatti il monitoraggio e l'elaborazione dei messaggi attraverso code e mediante l'utilizzo di backpressure[10], ove necessario. Inoltre lo scambio di messaggi è uno stile di comunicazione non bloccante, dunque fa sì che l'entità ricevente possa consumare le risorse quando è attiva, il che porta ad un minor sovraccarico del sistema.

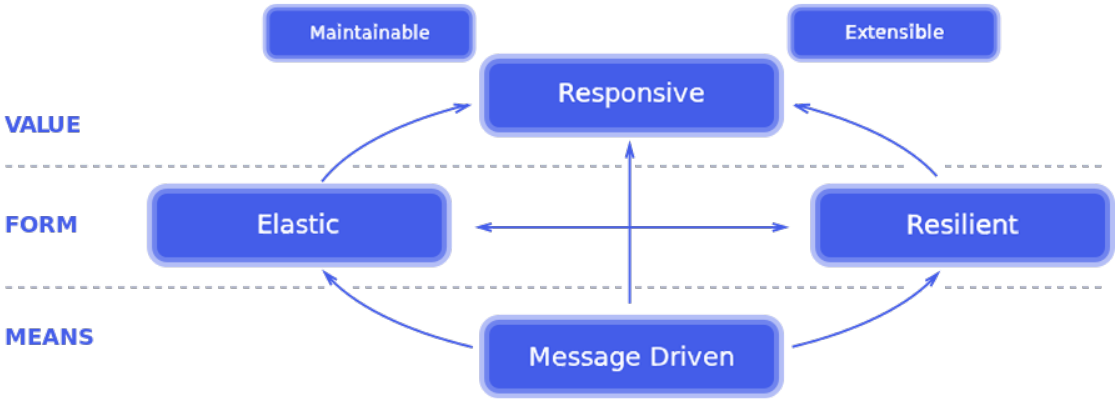


Figura 1.11: Gli elementi costitutivi principali di un sistema reattivo

L'ottemperanza alle direttive enunciate nel manifesto garantisce inoltre che il sistema risultante guadagni automaticamente caratteristiche importanti quali l'alto grado di manutenibilità e di estensibilità.

[10]La backpressure è una tecnica impiegata per risolvere il problema che si viene a creare nel momento in cui una sorgente di dati/eventi emette i suoi elementi ad una velocità superiore rispetto a quella che il suo sottoscrittore riesce a gestire. In questo caso la soluzione sta nel permettere all'osservatore di inviare un feedback alla sorgente (in modo tale che questa possa diminuire il flusso di dati in uscita) oppure nell'attuare strategie per la gestione dei dati in eccesso.

Capitolo 2

Linguaggio Kotlin

Nel 2011 *JetBrains*, azienda conosciuta per aver creato *IntelliJ IDEA*, un *IDE* (*Integrated Development Environment*) per *Java*, ha annunciato lo sviluppo del linguaggio di programmazione *Kotlin*; a detta della compagnia, si sarebbe posto come valida alternativa alla scrittura di codice in linguaggi come Java e *Scala*, che girano su *Java Virtual Machine* (*JVM*). Kotlin è cresciuto rapidamente, suscitando sempre maggior interesse e imponendosi, sei anni dopo la sua nascita, come il linguaggio di programmazione di riferimento per le applicazioni del sistema operativo mobile più diffuso al mondo. Nell'ottobre del 2017 infatti, in occasione dell'uscita del nuovo IDE *Android Studio 3.0*, Google ha deciso di scegliere Kotlin come linguaggio ufficialmente supportato per lo sviluppo di app sul sistema operativo Android, affiancando Java e C/C++.

Presenta sintassi concisa, interoperabilità con Java ed altre caratteristiche che verranno via via descritte all'interno di questo capitolo.

2.1 Perché Kotlin

Java è un linguaggio di programmazione robusto e ben testato nel corso del tempo ed è attualmente fra i più utilizzati al mondo[1]. Tuttavia, da quando è stato rilasciato per la prima volta nel 1995, sono state numerose le nuove tecniche messe a punto per quanto riguarda le regole di buona programmazione e Java non sempre è riuscito ad adattarsi bene, come invece hanno dimostrato altri linguaggi più moderni, nati di recente. Kotlin è uno di questi: un linguaggio che cerca di trarre ispirazione dalle migliori tecniche di programmazione e progettazione.

Kotlin non si configura solo come un linguaggio moderno per scrivere codice da eseguire su *Java Virtual Machine*, ma è anche un linguaggio multipiattaforma: è infatti in grado di essere compilato in codice *JavaScript* o direttamente in nativo tramite la struttura di compilazione *LLVM* [16] ed essere, naturalmente, anche eseguito su dispositivi Android.

[1]Per maggiori informazioni visitare: `http://archive.fo/h3T7g`

Non per ultimo, Kotlin si propone di essere un linguaggio di programmazione conciso, sicuro, "tool-friendly"[2] e di diminuire le linee di codice di circa il 40% rispetto a quelle necessarie per la scrittura in Java[3].

2.1.1 Diffusione, crescita e pareri nel mondo

La comunità interessata a Kotlin è in espansione: esistono attualmente oltre 100 gruppi di utenti in tutto il mondo r numerosi talk e conferenze. In figura 2.1 sono riportati i talk su Kotlin attualmente riconosciuti in giro per il mondo[4]:

Figura 2.1: Kotlin talks in tutto il mondo (primo trimestre 2018)

Per quanto riguarda invece la crescita di Kotlin: su GitHub ha ormai superato abbondantemente 25 milioni di linee di codice (figura 2.2), con crescita pressoché esponenziale, mentre su Stack Overflow è etichettato come uno dei linguaggi di programmazione con la crescita più veloce, nonché uno tra i più apprezzati in assoluto[5].

[2]Per maggiori informazioni visitare: `https://kotlinlang.org/`
[3]Per maggiori informazioni visitare: `https://kotlinlang.org/docs/reference/faq.html`
[4]Per maggiori informazioni visitare: `http://kotlinlang.org/community/talks.html`
[5]Per maggiori informazioni visitare: `http://archive.is/P1A0E`

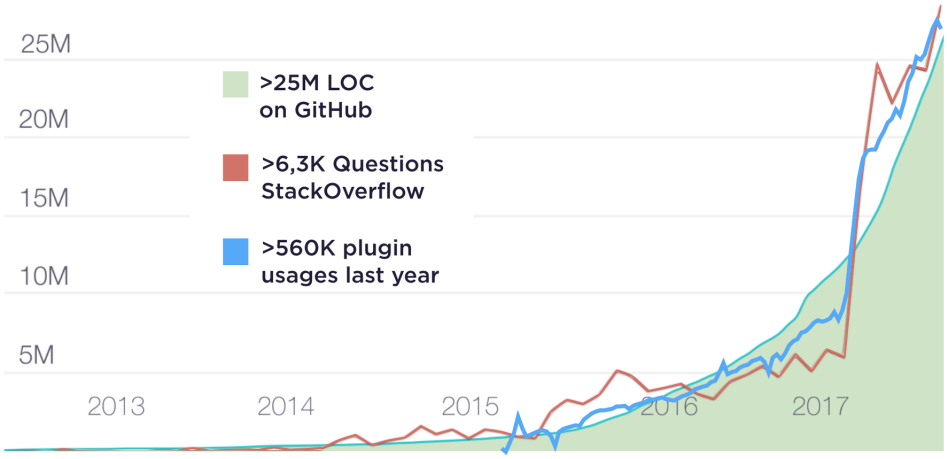


Figura 2.2: Crescita di Kotlin nel tempo (in ascissa l'anno, in ordinata il numero di linee di codice espresse in milioni)

In figura 2.3, sono mostrate le percentuali di gradimento per ognuna delle caratteristiche più rilevanti di Kotlin. Non è un caso che la feature più votata sia la *null safety*, essa è difatti molto richiesta dagli sviluppatori ma assente in molti altri linguaggi di programmazione.

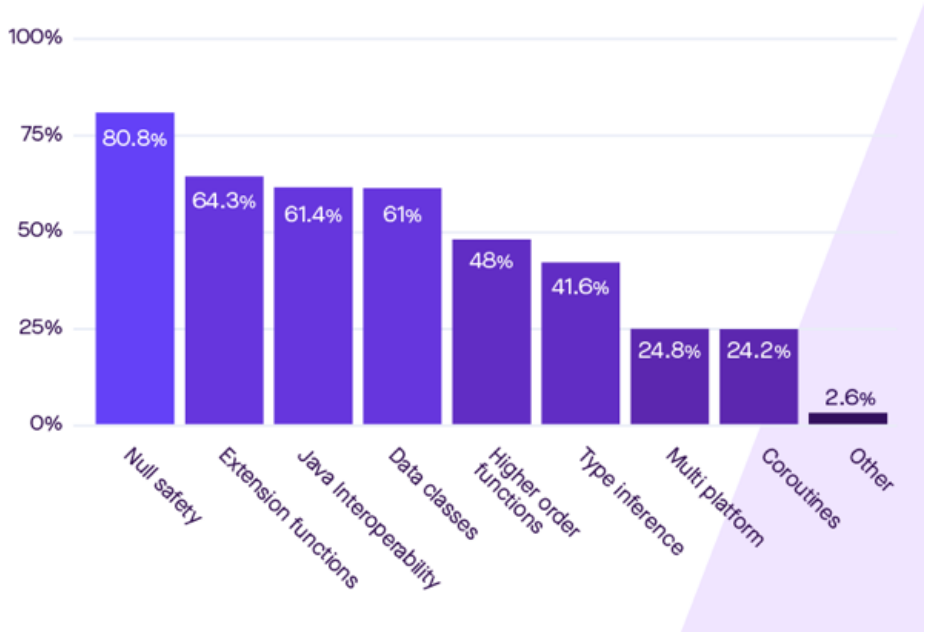


Figura 2.3: Le caratteristiche di Kotlin che vengono maggiormente apprezzate dagli sviluppatori

2.2 Caratteristiche fondamentali

Kotlin è un linguaggio di programmazione a tipizzazione statica che possiede sia i costrutti tipici del paradigma di programmazione ad oggetti, sia alcuni di quelli del paradigma di programmazione funzionale, permettendo perciò l'utilizzo di entrambi gli stili, anche contemporaneamente.

Fu creato dietro ispirazione di linguaggi già esistenti, quali *Scala*, *C#*, *Ceylon*, *Groovy*, e *Java*. Per questo motivo, e per essere stato progettato in modo tale che sia semplice e intuitivo da imparare, possiede una buona curva di apprendimento per qualunque programmatore (specialmente se quest'ultimo mostra dimestichezza con i linguaggi sopra elencati), che può iniziare a padroneggiare le sue basi dopo soli pochi giorni di studio. Tuttavia, per poter utilizzare le caratteristiche di dettaglio e le funzionalità più avanzate di Kotlin è necessario un maggior impegno e un tempo superiore; nonostante ciò, non viene considerato un linguaggio particolarmente complicato.

Di seguito vengono elencate le principali caratteristiche di Kotlin e messe a confronto, ove necessario, con quelle presenti in Java. Tuttavia è bene ricordare che in questa sezione si accennano soltanto le proprietà basilari, per cercare di dare un'idea iniziale e generale del linguaggio; altre importanti caratteristiche di Kotlin verranno esposte in seguito all'interno della tesi.

- **Concisione ed espressività**

  È possibile scrivere molto utilizzando poco codice (in media circa il 40% in meno rispetto a quello necessario in Java[6]).

  Con Kotlin è molto semplice evitare sezioni di codice verboso perché i pattern più comuni sono supportati di default direttamente dal linguaggio. Ad esempio, in Java, se si vuole creare una *data class* (o *POJO*, *Plain Old Java Object*, cioè un oggetto contenente i campi e i soli metodi per accedere e settare tali campi), bisogna scrivere (o almeno generare) il codice presente nel listato 2.1:

```java
public class Product {
    private long id;
    private String name;
    private String description;

    public long getId() {
        return this.id;
    }

    public void setId(final long id) {
        this.id = id;
    }

    public String getName() {
```

JAVA

[6]Per maggiori informazioni visitare: `https://kotlinlang.org/docs/reference/faq.html`

```
15          return this.name;
16      }
17
18      public void setName(final String name) {
19          this.name = name;
20      }
21
22      public String getDescription() {
23          return this.description;
24      }
25
26      public void setDescription(final String description) {
27          this.description = description;
28      }
29
30      @Override
31      public String toString() {
32          return "Product{" +
33                  "id = " + this.id +
34                  ", name = " + this.name +
35                  ", description = " + this.description + "}";
36      }
37  }
```

Listato 2.1: Creazione di un POJO in Java

In Kotlin, invece, creare lo stesso *POJO* con metodi *getters*, *setters*, *equals()*, *hashCode()*, *toString()* e *copy()* necessita molto meno codice, listato 2.2:

```
1  data class Product(                                          KOTLIN
2      var id: Long,
3      var name: String,
4      var description: String)
```

Listato 2.2: Creazione di un POJO in Kotlin

Ora vengono mostrate invece le linee di codice necessarie a creare un *singleton* in Kotlin (listato 2.3)[7]:

```
1  object ThisIsASingleton {
2      val favoriteLanguage: String = "Kotlin"
3  }
```

Listato 2.3: Creazione di un singleton in Kotlin

Ed infine, per dimostrare ancora una volta come Kotlin sia un linguaggio di programmazione conciso ed espressivo, di seguito viene mostrata l'operazione

[7]Per maggiori informazioni visitare: `https://kotlinlang.org/`

di filtraggio di una lista utilizzando le *lambda expression* (listato 2.4, pagina successiva).

```
1  val list = listOf(-2, -1, 0, 1, 2, 3)
2  val positiveNumbers = list.filter { it > 0 }
```

Listato 2.4: Filtraggio di una lista in Kotlin, con lambda

- **Null-safety**

  Quando si utilizza Java, buona parte del codice viene scritto a scopo 'difensivo'; è opportuno difatti controllare correttamente che un oggetto non sia nullo prima di impiegarlo, altrimenti si potrebbe incorrere in una `NullPointerException`.

  Kotlin, come altri linguaggi moderni, è *null-safe*: i tipi in Kotlin definiscono infatti se un oggetto può essere nullo (attraverso l'uso del suffisso punto interrogativo `?`[8]) oppure se non può esserlo (assenza del suffisso `?`). Inoltre Kotlin è tipizzato staticamente, questo significa che tutti i tipi presenti in ogni sua espressione sono conoscibili a tempo di compilazione: indicati esplicitamente dal programmatore oppure inferiti dal compilatore stesso. In questo modo gli eventuali errori di tipo vengono intercettati prima che il codice venga eseguito. Di seguito, nel listato 2.5, alcuni esempi[9]:

```
1  var output: String   // output can't be null
2  output = null   // Compilation error
3
4  var artist: Artist? = null   // artist can be null
5  artist.print()   // Compilation error, artist could be null
6  artist?.print()   // Will print only if artist is not null
7
8  var name = "John"   // Inferred type is String
9  name = "Cathy"
10 name = 3   // Error, because name type is String
```

Listato 2.5: Null-safety in Kotlin

Un'altra funzionalità di Kotlin, che aiuta a renderlo più solido e conciso, è lo *auto-cast* (listato 2.6):

```
1  fun calculateTotal(obj: Any) {
2      if (obj is Invoice) {
3          obj.calculateTotal()   // Cast is not necessary
```

[8] Il suffisso punto interrogativo `?` non va confuso con l'*operatore di chiamata sicura* `?.` o con l'*operatore Elvis* `?:` (questi ultimi verranno descritti nella prossima sezione).

[9] Per maggiori informazioni visitare: `https://kotlinlang.org/`

```
4         }
5     }
```

Listato 2.6: Auto-cast in Kotlin

- **Interoperabilità**

  É possibile continuare ad utilizzare librerie e codice scritto in Java, poiché esso è interoperabile con Kotlin. Si può dunque facilmente richiamare codice Kotlin da codice Java e viceversa, e creare progetti con i due linguaggi utilizzati contemporaneamente. Infine, è presente anche un convertitore Java-to-Kotlin automatizzato integrato negli IDE IntelliJ IDEA ed Eclipse, semplificando la migrazione del codice esistente.

- **Approccio funzionale**

  Kotlin, come già accennato, è fondamentalmente un linguaggio orientato agli oggetti, non un puro linguaggio funzionale; tuttavia, come molti altri linguaggi moderni, utilizza molti concetti dalla programmazione funzionale (come per esempio espressioni lambda, i *function types* e le funzioni di ordine superiore) per risolvere alcuni problemi in maniera molto più semplice.

  Le funzioni in Kotlin sono di 'prima classe' (*first-class*), ciò significa che possono essere memorizzate all'interno di variabili o in strutture dati, passate come argomenti o ritornate da parte di altre funzioni di grado superiore. É possibile quindi gestire le funzioni in tutti i modi con i quali è possibile gestire gli altri tipi di valori. Kotlin è perciò usabile da chi abitualmente programma in maniera funzionale ed è una buona scelta per iniziare ad esplorare questo paradigma di programmazione.

- **Tool-friendly**

  Kotlin è supportato da tutti i principali IDE Java tra cui *IntelliJ IDEA*, *Android Studio*, *Eclipse* e *NetBeans*.

- **Ottima curva di apprendimento**

  Come detto, Kotlin è dotato di un plugin che permette la conversione automatica da Java a Kotlin; è però presente anche una guida[10] che percorre le funzionalità principali del linguaggio attraverso una serie di esercizi interattivi. Tendendo conto anche delle sue somiglianze con Java, Kotlin risulta avere una curva di apprendimento ottima per quasi qualunque programmatore avvezzo a Java o Scala.

[10]Per maggiori informazioni visitare: `http://archive.is/8DOpN`

2.3 Sintassi e funzionalità di Kotlin

In questa importante sezione verranno descritte le principali funzionalità di Kotlin e la sua sintassi, partendo dalle basi del linguaggio, sino ad arrivare ad argomenti più complessi. Si tralascia di esporre alcuni concetti di base già presenti in Java, assumendo che il lettore ne sia in generale a conoscenza.

2.3.1 Hello world

La modalità più rapida per provare il linguaggio senza necessità di installare alcun software è quella di utilizzare *Kotlin Koans Online*[11]: è infatti possibile eseguire codice utilizzando le implementazioni JVM Kotlin o JavaScript e scegliere la versione di Kotlin desiderata fra le differenti disponibili.

La funzione *main*, come per Java, rimane il punto di partenza per ogni applicazione Kotlin; questa funzione viene chiamata quando il programma si avvia (listato 2.7):

```
1 fun main(args: Array<String>) {
2     println("Hello world!")
3 }
```

Listato 2.7: Esempio di *Hello world* in Kotlin

Nelle applicazioni Android, tuttavia, la funzione *main* non viene mai scritta direttamente dal programmatore; essa viene infatti chiamata implicitamente dal framework, quindi in questo specifico caso non può essere utilizzata in maniera esplicita per eseguire un *Hello world* in Kotlin. Nel successivo listato 2.8 viene dunque mostrato il codice Kotlin necessario per far apparire un messaggio di saluto all'interno di una `TextView` nell'activity principale:

```
1 import kotlinx.android.synthetic.main.activity_main.textView
2
3 class MainActivity : AppCompatActivity() {
4     override fun onCreate(savedInstanceState: Bundle?) {
5         super.onCreate(savedInstanceState)
6         setContentView(R.layout.activity_main)
7         textView.text = "Hello world!"
8     }
9 }
```

Listato 2.8: Esempio di *Hello world* in Kotlin su Android

Come si può da subito notare, una caratteristica ben visibile che contraddistingue questo linguaggio da Java è sicuramente l'assenza del punto e virgola come

[11] Per maggiori informazioni visitare: `https://try.kotlinlang.org/`

terminatore delle istruzioni: in Kotlin non è difatti più necessario, a meno che si vogliano concatenare più istruzioni su una stessa linea di codice.

A parte quest'evidente differenza di base, ora non è necessario comprendere perfettamente le altre caratteristiche e meccanismi mostrati nei due esempi precedenti (utili soprattutto per farsi un'idea generale della sintassi di Kotlin), poiché verranno spiegati nel dettaglio successivamente.

2.3.2 Variabili

In Kotlin tutto è rappresentato da oggetti, di conseguenza anche le variabili possiedono sempre un tipo oggetto e mai tipi primitivi[12], poiché il linguaggio non ne prevede (diversamente da Java); questo rende il codice meno complesso.

Le variabili in Kotlin possono essere dichiarate utilizzando una delle due parole chiave: `var` oppure `val`.

La prima indica un *riferimento mutabile* che può subire modifiche e aggiornamenti dopo l'inizializzazione; è l'equivalente di una 'normale' (cioè non finale) variabile in Java. Dunque se una variabile necessita di cambiare valore nel tempo, essa deve necessariamente essere dichiarata usando la parola chiave `var`.

La seconda invece, `val`, indica un *riferimento immutabile* (read-only) che *non* può essere riassegnato una volta inizializzato; è l'equivalente di una variabile finale (dichiarata utilizzando il modificatore `final`) in Java. Utilizzare una variabile di tipo `val` è utile e sempre consigliato ove possibile, in quanto assicura che la stessa non venga mai modificata erroneamente. Risulta inoltre vantaggioso impiegare variabili di questa tipologia quando si lavora con più thread: poiché promuovono immutabilità, semplificando la sincronizzazione degli accessi [22].

Un semplice esempio esplicativo nel seguente listato 2.9:

```
1  var animal = "cat"
2  animal = "lion"   // Correct
3
4  val craft = "painter"
5  craft = "sculptor"   // Compilation error: val cannot be reassigned
```

Listato 2.9: Esempio di variabili dichiarate `var` e `val`

Da notare che quando si utilizza `val` non si è più in grado di cambiare il riferimento che punta ad un particolare oggetto, ma questo non sempre significa che non si possano modificare le proprietà interne dell'oggetto stesso. La parola chiave `val` non può infatti garantire che l'oggetto referenziato sia immutabile. Per chiarire il concetto, viene mostrato un esempio nel listato 2.10:

[12] In realtà alcuni tipi oggetto particolari, quali quelli per indicare numeri, caratteri e booleani, vengono rappresentati (se non nulli) tramite i rispettivi tipi primitivi a tempo d'esecuzione; tuttavia il processo è del tutto trasparente e l'utente li può (li deve) sempre trattare come ordinarie classi: la conversione è implicita e scaricata sul compilatore.

```
1 val list = mutableListOf("a", "b", "c")
2 list.remove("a")   // List modified, correct
3 list = mutableListOf("d", "e")   // Reference modified, incorrect!
```

Listato 2.10: Variabile `val`, modifica dello stato dell'oggetto e del riferimento

Se è invece necessario che un oggetto non venga in alcun modo modificato, allora si deve utilizzare un riferimento immutabile e un oggetto immutabile: al fine di favorire l'uso di strutture dati immutabili, la libreria standard di Kotlin contiene un equivalente immutabile per ogni collezione (`List` per `MutableList`, `Map` per `MutableMap`, e così via).

2.3.3 Tipi di dato

I tipi di dato presenti in Kotlin si ispirano a quelli di Java, sebbene si possano facilmente rilevare alcune differenze importanti che meritano di essere sottolineate. Come già specificato precedentemente, una differenza risiede nel fatto che il type system di Kotlin non prevede tipi di dato primitivi (posseduti invece da Java), poiché il linguaggio abbraccia una filosofia totalmente a oggetti; tuttavia, questa non è l'unica differenza che contraddistingue Kotlin dal suo predecessore (le rimanenti disparità verranno presto evidenziate).

In questa sottosezione vengono elencati e descritti separatamente i tipi di dato di base presenti in Kotlin, fondamentali per la rappresentazione rispettivamente di numeri, caratteri, booleani, vettori e stringhe.

Numeri

I tipi di dato per rappresentare i numeri in Kotlin (tabella 2.1) sono simili a quelli presenti in Java, tranne per il fatto che non esistono implicite conversioni fra tipi compatibili e che `Char` non può contenere numeri (esempi nel listato 2.11).

Type	Bit width
Byte	8
Short	16
Int	32
Long	64
Float	32
Double	64

Tabella 2.1: Tipi di dato per rappresentare i numeri in Kotlin

```
1  val s: Short = 3
2  val i1: Int = s   // Error, incorrect type conversion!
3  val i2: Int = s.toInt()   // Correct type conversion
4
5  val b: Char = 1   // Error, incorrect!
```

Listato 2.11: Alcuni esempi con tipi di dato numerici in Kotlin

Un'altra differenza con Java sta nel fatto che in Kotlin non sono supportate le costanti letterali per i valori in ottale.

Caratteri

I caratteri in Kotlin sono rappresentati dal tipo di dato `Char` che, come mostrato nel precedente listato 2.11, non può contenere numeri.

Come per Java, i caratteri devono essere essere delimitati necessariamente da apici semplici e quelli speciali devono iniziare con un *backslash*. Le *sequenze di escape* supportate sono \t, \b, \n, \r, \', \", \\, \$ e \u.

Nel prossimo listato 2.12 sono presenti alcuni esempi:

```
1  val c1: Char = 'z'
2  val c2: Char = "z"   // Incorrect: "z" is a String!
3
4  val i: Int = '3'.toInt()   // Correct, '3' is a Char
5
6  val c3: Char = '\uFF00'   // Correct use of escape sequence
```

Listato 2.12: Alcuni esempi con il tipo di dato `Char` in Kotlin

Booleani

I booleani in Kotlin sono rappresentati dal tipo di dato `Boolean`. Questo può assumere come prevedibile soltanto sue possibili valori: `true` o `false`.

Sono inoltre supportate la operazioni standard built-in che sono generalmente disponibili anche in altri moderni linguaggi di programmazione:

- `||` : rappresenta l'OR logico. Restituisce `true` quando uno dei due predicati ritorna `true`.
- `&&` : rappresenta l'AND logico. Restituisce `true` quando entrambi i predicati ritornano `true`.
- `!` : operatore di negazione. Restituisce `true` per `false` e viceversa.

Come accade in Java, `||` e `&&` sono valutati in maniera *lazy* (rispettivamente tramite *lazy disjunction* e *lazy conjunction*).

Vettori

In Kotlin, i vettori (array) vengono rappresentati dalla classe `Array`, che possiede i metodi `get` e `set` per prelevare ed impostare valori e il campo `size` per ottenere la lunghezza del vettore.

Diversamente da Java, gli array in Kotlin sono invarianti; questo significa che in Kotlin non è permesso assegnare un `Array<Int>` a un `Array<Any>`, per prevenire possibili errori a tempo di esecuzione. Si possono definire in Kotlin array covarianti con `Array<out Any>` (che corrisponde ad `Array<? extends Object>` in Java) oppure array controvarianti con `Array<in String>` (che corrisponde ad `Array<? super String>` in Java).

Per creare un vettore può essere utilizzata la funzione di libreria `arrayOf()`, a cui passare come argomenti i valori da inserire, oppure direttamente il costruttore di `Array` che prende in ingresso la lunghezza e una funzione che restituisce il valore di ogni elemento del vettore per ogni dato indice. Alcuni esempi nel listato 2.13:

```
1  val array = arrayOf(1, 2, 3, 4)   // Inferred type: Array<Int>
2  val array2: Array<Long> = arrayOf(1, 2, 3, 4)
3
4  // Creates an Array<String> with values ["0", "3", "6", "9", "12", "15"]
5  val array3 = Array(6, { i -> (i * 3).toString() })
```

Listato 2.13: Alcuni esempi di creazione di vettori in Kotlin

Stringhe e string templates

Il comportamento delle stringhe in Kotlin è simile a quello visibile in Java, con alcuni miglioramenti.

Le stringhe in Kotlin si indicano con il tipo di dato `String` e sono, come in Java, *immutabili*.

É possibile accedere agli elementi delle stringhe tramite l'*indexing operator* (per esempio `str[i]`), nello stesso modo in cui si accede agli elementi di un vettore, oppure tramite l'iterazione all'interno di un ciclo. La concatenazione di più stringhe avviene come in Java tramite l'operatore `+` e funziona anche quando è necessario concatenare stringhe e altri tipi di valori.

Nonostante sia presente l'operatore di concatenazione, è spesso preferibile utilizzare i cosiddetti *string templates*: è possibile posizionare una variabile all'interno di una stringa tramite l'uso del carattere `$` che funge da segnaposto. Durante l'interpolazione, i segnaposto vengono rimpiazzati con i rispettivi valori. Questo meccanismo rende il codice più ordinato, evitando di dividere la stringa in porzioni ogniqualvolta occorra inserire il valore di una variabile o di un'espressione.

Nel listato 2.14 vengono mostrati alcuni esempi:

```
1  val str = "Hello"
2  val c = str[1]   // Inferred  type: Char
3  for (i in str) {
4      // ...
5  }
6
7  val str2 = "alfa" + 1 + "beta" + 2   // str2 = "alfa1beta2"
8
9  val i = 30
10 println("i = $i")   // Prints "i = 30"
11
12 val str3 = "abcde"
13 println("$str3.length is ${str3.length}")   // Prints "abcde.length is 5"
```

Listato 2.14: Alcuni esempi di utilizzo delle stringhe e *string templates* in Kotlin

2.3.4 Inferenza di tipo

Come già accennato negli esempi precedenti, in Kotlin quando la dichiarazione e l'inizializzazione di una variabile avvengono sulla stessa linea di codice, è possibile omettere l'indicazione del tipo della variabile. Questo meccanismo, non presente in questi termini in Java, viene chiamato *inferenza di tipo*: il compilatore inferisce automaticamente il tipo dal contesto, perciò non è necessario che il programmatore lo specifichi esplicitamente.

È possibile osservare inferenza in molteplici esempi precedentemente mostrati: a linee 1 e 2 del listato 2.4, a linea 8 del listato 2.5, a linee 1 e 4 del listato 2.9 ed infine a linea 1 del listato 2.10.

Se tuttavia si volesse assegnare un valore intero ad una variabile contenente una stringa, bisognerebbe specificare un tipo in comune tra `String` e `Int`: in questo caso si dovrebbe optare per `Any` (esempio nel listato 2.15).

```
var name: Any = "Sara"
name = 15   // Allowed
```

Listato 2.15: Utilizzo del tipo `Any`

`Any` è l'equivalente del tipo `Object` di Java, rappresenta la radice della gerarchia dei tipi di Kotlin, di conseguenza tutte le classi ereditano da esso (pertanto anche le classi `String` e `Int`).

Come si può notare dal prossimo listato 2.16, l'inferenza di tipo non si limita soltanto ai tipi di dato di base, ma funziona egregiamente anche con funzioni e tipi composti quali coppie di valori e mappe:

```
fun sum(a: Int, b: Int) : Int {
    return a + b
}

// Inferred type: Int
val total = sum(30, 40)

// Inferred type: Pair<Int, String>
val pair = total to "Seventy"

// Inferred type: Pair<Pair<Int, String>, Float>
val multiPair = total to "Seventy" to 70.0f

// Inferred type: Map<Int, String>
val map = mapOf(total to "Seventy", 90 to "Ninety")

// Inferred type: Map<Int, Any>
val superMap = mapOf(total to "Seventy", 90 to 90.0f)
```

Listato 2.16: Applicazioni d'inferenza di tipo in Kotlin

2.3.5 Strict null safety

Come probabilmente è stato sperimentato nel tempo da molti programmatori Java, una delle sorgenti d'errore più diffuse è il non corretto controllo dei valori nulli, con conseguente lancio dell'insidiosa *NullPointerException*.

Per evitare questo genere di problemi alcuni linguaggi moderni, compreso Kotlin, possiedono un meccanismo di sicurezza dalla nullabilità inserito direttamente all'interno del proprio type system. In questo modo è possibile scrivere codice molto più sicuro e convertire gli errori a tempo di esecuzione in errori a tempo di compilazione; un vantaggio non da poco. Il meccanismo viene chiamato *strict null*

safety (o più semplicemente *null safety*) e funziona poiché il type system di Kotlin distingue tra i riferimenti che possono contenere valori nulli (*nullable references*) e quelli che invece non possono (*non-nullable references*). Di default i tipi non possono contenere valori nulli e ciò viene reso possibile solamente se esplicitamente indicato dal programmatore. Un esempio è quello mostrato da riga 1 a 6 del precedente listato 2.5.

Operatore di chiamata sicura

L'*operatore di chiamata sicura* (*safe call operator*) si indica con i caratteri `?.` e si comporta nel seguente modo: prima di tutto compie in automatico un controllo sulla nullabilità della parte a sinistra dell'operatore, se l'esito risulta essere positivo allora restituisce `null`, altrimenti (valore non nullo) esegue e restituisce il risultato dell'espressione sulla parte destra.

La sintassi di questo operatore è la seguente:

```
object?.method
```

Nei seguenti listati 2.17 e 2.18 vengono mostrati due esempi a confronto (il primo in codice Java, il secondo in Kotlin) su Android:

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Correct compilation but possible error at runtime!
    final boolean isSwitchedOn = savedInstanceState.getBoolean("switch");
}
```

JAVA

Listato 2.17: Possibile errore a tempo di esecuzione in Java

Sebbene il codice Java compili correttamente, la chiamata al metodo `getBoolean` sull'oggetto `savedInstanceState` può provocare un errore a tempo di esecuzione, determinando il lancio di una `NullPointerException`. Questo accade a causa della dimenticanza dei controlli sulla nullabilità dell'oggetto `savedInstanceState`.

```kotlin
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val isSwitchedOn: Boolean? = savedInstanceState?.getBoolean("switch")
}
```

KOTLIN

Listato 2.18: Operatore di chiamata sicura in Kotlin

Nell'esempio in Kotlin, invece, questo problema viene risolto utilizzando l'operatore di chiamata sicura: se `savedInstanceState` è nullo, allora viene restituito `null`, altrimenti viene chiamato il metodo `getBoolean` e restituito il risultato.

Da tenere a mente tuttavia che i riferimenti potenzialmente nulli possono restituire un valore nullo, perciò è necessario (il compilatore Kotlin lo impone) che il risultato dell'espressione sia di tipo `Boolean?` (specificato esplicitamente o inferito automaticamente), come indicato nell'esempio.

Questo operatore è utile anche durante le chiamate a catena. Se per esempio Bob, un dipendente, può essere assegnato a un dipartimento (oppure no), il quale a sua volta può possedere un altro dipendente come capo dipartimento, allora per ottenere il nome del capo dipartimento di Bob (se presente), è possibile scrivere quanto segue[13] nel listato 2.19:

```
1 val name = bob?.department?.head?.name
```

Listato 2.19: Operatore di chiamata sicura utilizzato in una catena di chiamate

La precedente espressione restituisce quindi `null` soltanto se rileva che un oggetto è nullo; in questo modo si risparmia di controllare con molteplici `if` la nullabilità o meno di ogni oggetto.

Operatore elvis

L'*operatore elvis* (*Elvis operator*) viene rappresentato per mezzo dei due caratteri `?:` e si comporta nella maniera seguente: se l'operando di sinistra *non* è nullo, allora l'operatore di elvis lo restituisce come risultato, altrimenti restituisce il secondo operando.

La sintassi di questo operatore è:

```
first operand ?: second operand
```

Nel listato 2.20 ne viene mostrato un esempio pratico:

```
1 override fun onCreate(savedInstanceState: Bundle?) {
2     super.onCreate(savedInstanceState)
3     val isSwitchedOn: Boolean = savedInstanceState?.getBoolean("switch") ?: false
4 }
```

Listato 2.20: Operatore elvis in Kotlin

L'operatore elvis nell'esempio restituisce il valore dell'espressione `savedInstanceState?.getBoolean("switch")` se `savedInstanceState` *non* è nullo, altrimenti restituisce `false`. Questo operatore permette quindi di specificare un valore di

[13] Per maggiori informazioni visitare: `http://archive.is/KtvGp`

default da restituire solo nel caso l'operando di sinistra non produca un risultato idoneo (cioè non nullo).

Risulta inoltre importante sottolineare il fatto che in questo caso la variabile `isSwitchedOn` è di tipo `Boolean`: non è infatti più necessario specificarla come `Boolean?`, in quanto l'operatore di elvis impedirà sempre che possa essere `null`.

Questo operatore può essere utilizzato ovviamente anche congiuntamente a catene di chiamate, ne è un buon esempio il listato 2.21:

```
1 val name = bob?.department?.head?.name ?: "Jack"
```

Listato 2.21: Operatore elvis utilizzato insieme a una catena di chiamate

Operatore di asserzione non nulla

L'*operatore di asserzione non nulla* (*not-null assertion operator*) viene indicato con i tre caratteri `!!.` e compie implicitamente un cast di una variabile potenzialmente nulla ad una non nulla. Il suo utilizzo è però fortemente sconsigliato, poiché può essere fonte di errori a runtime e portare ad una `NullPointerException`.

Se il programmatore utilizza questo operatore su una variabile, significa che sta dichiarando al compilatore che essa contiene effettivamente un valore non nullo; perciò non spetta più a quest'ultimo il giudizio finale sulla fondatezza o meno della dichiarazione e la responsabilità di tale azione ricade totalmente sul programmatore. Un semplice esempio nel prossimo listato 2.22:

```
1 override fun onCreate(savedInstanceState: Bundle?) {
2         super.onCreate(savedInstanceState)
3         val isSwitchedOn: Boolean = savedInstanceState!!.getBoolean("switch")
4 }
```

Listato 2.22: Operatore di asserzione non nulla in Kotlin

La sintassi di questo operatore è dunque:

```
object!!.method
```

Nel caso in cui la variabile `savedInstanceState` nell'esempio contenga un valore non nullo, allora tutto procederà correttamente; in caso contrario, invece, l'applicazione andrà incontro a crash e verrà lanciata una `NullPointerException`.

Si può notare come questo comportamento sia effettivamente simile a quello adottato da Java, ma con una differenza di fondo: in Java l'accesso ad oggetti potenzialmente non nulli e senza controllo di nullabilità è il comportamento predefinito, mentre in Kotlin questo va forzato intenzionalmente.

Operatore di cast sicuro

A volte chiamato anche *nullable cast operator*, l'*operatore di cast sicuro* (*safe cast operator*) si indica con i tre caratteri `as?` e agisce nel modo seguente: prova ad eseguire un cast di un valore ad uno specifico tipo e restituisce `null` se il procedimento non può essere attuato. La sintassi di questo operatore è:

```
object as? type
```

Nel seguenti listati 2.23 e 2.24 vengono mostrati due esempi di cast, il primo scritto in Java e il secondo in Kotlin:

```java
final int aInt = (int) number;   /* Possible ClassCastException */
```
JAVA

Listato 2.23: Cast non sicuro in Java

Nell'esempio in Java, in caso di insuccesso dell'operazione di cast il programma termina e viene tirata un'eccezione del tipo *ClassCastException*. Kotlin, invece, risolve il problema e riduce il codice necessario alla sua risoluzione facendo uso dell'operatore di cast sicuro:

```kotlin
val aInt: Int? = number as? Int
```
KOTLIN

Listato 2.24: Operatore di cast sicuro in Kotlin

Nell'esempio in Kotlin, l'operatore restituisce `null` in caso di insuccesso della conversione di tipo, altrimenti restituisce il risultato non nullo derivante dal cast. Ad ogni modo questo approccio non presenta alcun rischio, al contrario dell'esempio scritto in Java.

2.3.6 Flusso di controllo

Operatore when

L'operatore `when` in Kotlin sostituisce il costrutto `switch...case` presente in Java e lo migliora, offrendo un metodo per scrivere codice più conciso e leggibile.
L'operatore `when` compara il suo argomento con tutti i rami presenti all'interno del corpo finché non ne trova uno la cui condizione di uguaglianza risulti soddisfatta. Il comportamento è simile a quello presente in Java, sebbene in questo caso non si debba specificare l'istruzione `break` per ogni ramo. Il blocco `else` viene invece valutato solo nel caso in cui tutti i precedenti rami non siano soddisfatti.

Questo operatore può essere utilizzato in due diversi modi: come *espressione* o come *istruzione*; nel primo caso viene assegnato un valore (presente nell'ultima linea di codice del blocco che risulta soddisfatto) a una variabile, mentre nel secondo caso non è previsto un valore di ritorno (se presente viene ignorato).

Nel prossimo listato 2.25 viene mostrato un semplice utilizzo di `when`:

```
val numPaws = when (animal) {   // 'when' is used as an expression
    "Cat" -> {
        // Code ...
        "Four paws"
    }
    "Ostrich" -> {
        // Code ...
        "Two paws"
    }
    else -> {
        // Code ...
        "Unknown number of paws"
    }
}

when (num) {   // 'when' is used as a statement
    1 -> print("num == 1")
    2 -> print("num == 2")
    else -> {
        print("num is neither 1 nor 2")
    }
}
```

Listato 2.25: Utilizzo dell'operatore `when` in Kotlin

Cicli ed espressioni di range

In Kotlin i cicli vengono generati tramite tre diversi costrutti: `for`, `while` e `do...while`. Si ritiene non si necessario spiegare il loro funzionamento poiché risulta essere identico a quello presente in Java e in molti altri linguaggi di programmazione.

I cicli in Kotlin possono iterare su un qualsiasi oggetto che preveda un *iteratore* (cioè che implementi l'interfaccia `Iterator`). É dunque possibile iterare su una stringa, una collezione, un range e ogni altra entità che possa essere rappresentata come una sequenza di elementi.

Mentre le stringhe e le collezioni sono concetti ormai piuttosto conosciuti e presenti anche in Java, probabilmente lo sono meno le *espressioni di range*, esistenti fin dalla prima versione di Kotlin. Esse servono a rappresentare e generare una sequenza di valori a partire dall'indicazione del primo e dell'ultimo. Un range viene specificato tramite l'*operatore doppio punto* (*double dots operator*), cioè attraverso l'uso dei due caratteri `..` ('dietro le quinte' il linguaggio utilizza la funzione `rangeTo()`), e l'uso dell'operatore `in`. Questo meccanismo fa sì che le espressioni di range risultino particolarmente utili per l'impostazione di cicli, sebbene possano essere impiegate anche in altri tipi di situazioni (salvati in variabili o valutati all'interno di condizioni *if*, per esempio).

Di seguito (listato 2.26) vengono presentati alcuni esempi di range utilizzati per impostare cicli in Kotlin:

```
1  for (i in 1..6) {
2      print(i)   // Prints "123456"
3  }
4
5  for (i in 1 until 6) {
6      print(i)   // Prints "12345", end element is excluded
7  }
8
9  for (i in 'c'..'h') {
10     print(i)   // Prints "cdefgh"
11 }
12
13 for (i in 4..1) {
14     print(i)   // Prints nothing
15 }
16
17 for (i in 4 downTo 1) {
18     print(i)   // Prints "4321"
19 }
20
21 for (i in 6 downTo 1 step 2) {
22     print(i)   // Prints "642"
23 }
```

Listato 2.26: Utilizzo di range all'interno di cicli in Kotlin

Da notare che i range sono incrementali di default in Kotlin (righe 13-15), perciò è necessario usare `downTo` (riga 17) nel caso in cui si voglia ottenere una sequenza di elementi decrescenti (dietro le quinte viene utilizzata la funzione `downTo()`). A riga 21, invece, `step` (corrisponde alla funzione `step()`) indica quanti elementi bisogna scorrere ad ogni iterazione; il valore di default di `step` è 1.

2.3.7 Classi e oggetti

Kotlin consente di dichiarare ed istanziare classi in modo più conciso di Java (listato 2.27):

```
1  class Cat   // The simplest class declaration, an empty class
2  val cat = Cat()   // The 'new' keyword is not required
```

Listato 2.27: Dichiarazione di una classe e creazione di una sua istanza in Kotlin

Come si può notare non è necessario indicare il modificatore di accesso, poiché in Kotlin `public` è la visibilità di default; inoltre, se si desidera dichiarare una classe vuota come nell'esempio precedente, è possibile anche omettere le parentesi graffe.

Costruttori

In Kotlin i costruttori si distinguono in *primari* e *secondari*. Una classe può possedere un solo costruttore primario (il quale viene specificato direttamente nell'intestazione della classe stessa) e uno o più costruttori secondari.

Come viene mostrato nel prossimo snippet (listato 2.28), se si utilizza un costruttore primario e si necessita di codice di inizializzazione, questo va inserito all'interno di un blocco preceduto della parola chiave `init`; esso verrà eseguito ad ogni nuova creazione di un oggetto della corrispondente classe.

```
1  class Cat(val name: String, val isHungry: Boolean) {
2      init {
3          println("Cat instance created (name: $name)")
4      }
5  }
6
7  val cat = Cat("Oscar", true)   // Prints "Cat instance created (name: Oscar)"
```

Listato 2.28: Costruttore primario in Kotlin

I costruttori secondari, invece, vengono dichiarati utilizzando la parola chiave `constructor` (listato 2.29). Se una classe ha già un costruttore primario, ogni secondario deve richiamarlo direttamente o indirettamente (attraverso un altro costruttore secondario) tramite l'uso della keyword `this`:

```
1  class Cat(val name: String, val isHungry: Boolean) {
2      var weight : Float? = null
3
4      constructor(name: String, isHungry: Boolean, weight: Float)
5                  : this(name, isHungry) {
6          this.weight = weight
7      }
8  }
9
10 val firstCat = Cat("Oscar", true)   // Uses primary constructor
11 val secondCat = Cat("Molly", false, 4.5f)   // Uses secondary constructor
```

Listato 2.29: Utilizzo dei costruttori in Kotlin

Proprietà

Le *proprietà* in Kotlin sostituiscono il concetto di campo e di accessori di Java, si possono specificare nell'intestazione di una classe oppure nel suo corpo e rappresentano gli attributi posseduti da tutti gli oggetti istanziati dalla classe stessa.

Per fare un esempio, di seguito vengono presentati due snippet di codice equivalenti (listati 2.30 e 2.31), il primo in Java e il secondo in Kotlin:

```java
public class Person {
    private String name;
    private int age;

    public Person(final String name, final int age) {
        this.name = name;
        this.age = age;
    }

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

    public String getName() {
        return this.name;
    }

    public void setAge(final int age) {
        this.age = age;
    }

    public int getAge() {
        return this.age;
    }
}
```

JAVA

Listato 2.30: Campi e metodi *setter* e *getter* in Java

```kotlin
class Person(var name: String, var age: Int)
```

KOTLIN

Listato 2.31: Utilizzo delle proprietà in Kotlin

Per poter interagire con le proprietà si utilizza la seguente sintassi (listato 2.32):

```kotlin
val person = Person("Mike", 45)
person.age = 50
println(person.name)
```

Listato 2.32: Settare e ottenere i valori delle proprietà in Kotlin

In Kotlin è possibile specificare getter e setter personalizzati per ogni proprietà (i cosiddetti *accessors*), seguendo la struttura sintattica mostrata:

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

Di seguito alcuni semplici esempi pratici (listato 2.33):

```
var counter = 0   // Property
    get() = field + 1
    set(value) {
        if (value >= 0) {
            field = value
        }
    }

fun sample() {
    println("Counter: $counter")   // Prints 'Counter: 1' (0 + 1)
    counter = 21
    println("Counter: $counter")   // Prints 'Counter: 22' (21 + 1)
}
```

Listato 2.33: Proprietà con getter e setter personalizzati

Data classes

Capita spesso che sia necessario creare classi il cui unico scopo è quello di immagazzinare dati. In Kotlin queste classi sono specificate tramite la parola chiave `data` (per gli esempi si rimanda ai già presentati listati 2.1 e 2.2).

Durante la dichiarazione di una *data class*, il compilatore Kotlin risparmia al programmatore la scrittura di codice ridondante e derivabile (necessario in Java), facendosi carico di generare automaticamente:

- i metodi `equals()` ed `hashCode()`;
- il metodo `toString()`;
- i metodi `componentN()` per accedere alle proprietà;
- il metodo `copy()`;
- i metodi getter e setter corrispondenti alle proprietà specificate nel costruttore primario.

L'unica restrizione all'utilizzo delle data class è quella di non poter essere contrassegnate con le keyword `sealed`, `inner` e `abstract`.

Sealed classes

Kotlin offre anche la possibilità di creare tipi di dato algebrici tramite *sealed classes* (utilizzando la parola chiave `sealed`), cioè classi che possiedono un numero limitato e ben definito di sottoclassi. In un certo senso condividono alcune similitudini teoriche con le *classi di enumerazioni* (cosiddette *enum*) in quanto entrambe possiedono un insieme ristretto e preciso di valori, nonostante le classi sealed abbiano la capacità di possedere sottoclassi istanziabili.

Altre caratteristiche importanti delle sealed classes sono l'obbligo di indicare le sottoclassi all'interno dello stesso file dentro il quale è presente anche la classe madre e l'impossibilità di essere istanziate direttamente (in quanto astratte).

Nel prossimo listato 2.34 viene proposto un esempio di classe sealed (con rispettive sottoclassi) e una sua applicazione tramite l'uso dell'operatore `when`:

```
1  sealed class Animal
2  class Cat : Animal()    // Subclass
3  class Penguin : Animal()    // Subclass
4  class Ostrich : Animal()    // Subclass
5
6  val animal : Animal = Penguin()
7
8  val animalName = when(animal) {    // Unnecessary 'else' branch
9      is Cat -> "Cat"
10     is Penguin -> "Penguin"
11     is Ostrich -> "Ostrich"
12 }
```

Listato 2.34: Creazione e utilizzo di una sealed class in Kotlin

Da evidenziare come non sia necessario (e anzi scorretto) inserire il ramo *else* all'interno del corpo del `when` (righe 8-12), in quanto sono già state coperte tutte le possibili alternative.

Singleton

Singleton è uno dei Design Pattern [11] più utilizzati. Assicura che una classe abbia sempre una sola istanza.

Nel caso di Kotlin, come per Scala, viene messa a disposizione la keyword `object` (cosiddetta *object declaration*) per la creazione veloce di classi singleton.

Nella stessa maniera seguita per la dichiarazione di variabili, anche quella di classi `object` non rappresenta un'espressione, dunque non può ovviamente essere utilizzata a destra di un'istruzione di assegnamento. Un esempio esplicativo è già stato mostrato nel listato 2.3.

2.3.8 Funzioni

Per quanto riguarda le funzioni in Kotlin, nel prossimo listato (2.4) ne viene mostrata la struttura principale:

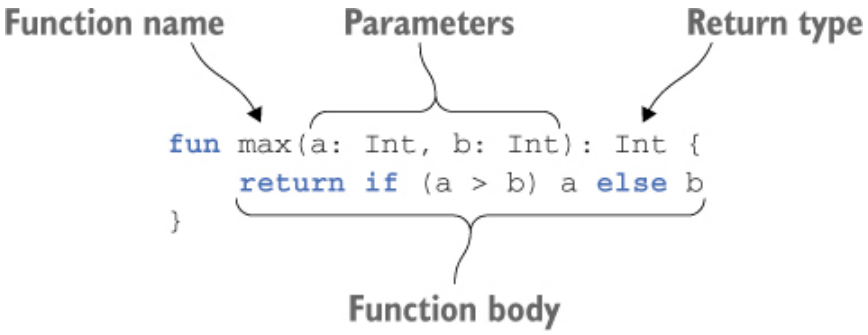


Figura 2.4: Struttura di una funzione in Kotlin

Come risulta semplice notare, per dichiarare una funzione in Kotlin è necessario anteporre la parola chiave `fun`, indicarne il nome, i parametri (e i rispettivi tipi) tra parentesi ed infine specificare il tipo di ritorno come ultima informazione (prima dell'apertura del corpo della funzione).

Kotlin permette al programmatore di specificare anche i valori predefiniti che devono assumere gli argomenti di una funzione nel caso vengano omessi in fase di chiamata alla funzione stessa. Questo permette di evitare l'*overloading*, tipicamente utilizzato in Java.

Nel caso in cui, tuttavia, ci si trovi innanzi ad una funzione i cui parametri di default precedono quelli *non* di default, i primi possono essere utilizzati soltanto se la funzione viene invocata con i nomi degli argomenti specificati (cosiddetti *named arguments*, listato 2.35).

```
1  private fun sum(num1: Int = 0, num2: Int = 0) : Int {
2      return num1 + num2
3  }
4
5  val total1 = sum(8, 5)    // num1 = 8, num2 = 5, total1 = 13
6  val total2 = sum(8)    // num1 = 8, num2 = 0 (default), total2 = 8
7  val total3 = sum()    // num1 = 0 (default), num2 = 0 (default), total3 = 0
8
9  private fun multiplication(num1: Int = 1, num2: Int) : Int {
10     return num1 * num2
11 }
12
13 val total4 = multiplication(6)    // Compilation error: num1 = 6, num2 = ?
14 val total5 = multiplication(num2 = 3)  // num1 = 1 (default), num2 = 3, total5 = 3
```

Listato 2.35: Utilizzo degli argomenti di default e *named arguments*

Quando una funzione restituisce una singola espressione, come nei casi presentati nel precedente esempio, è possibile omettere le parentesi graffe ed indicare il valore di ritorno dopo il simbolo `=` (omettendo perciò anche la keyword `return`); in questo caso, il tipo del valore di ritorno della funzione può essere inferito dal compilatore (listato 2.36):

```kotlin
fun sum(num1: Int, num2: Int) = num1 + num2
```

Listato 2.36: Esempio di funzione *single-expression*

Per ultimo, è importante ricordare che in Kotlin l'assenza di un valore di ritorno utile in una funzione si indica con il tipo `Unit` (contrariamente al tipo `void` presente in Java); quando utilizzato, permette di non specificare alcun ritorno esplicito nel corpo della funzione (listato 2.37).

```kotlin
fun printHello(name: String?) {   // `: Unit` is optional
    if (name != null) {
        println("Hello $name")
    } else {
        println("Hi there!")
    }
    // `return Unit` or `return` is optional
}
```

Listato 2.37: Esempio di utilizzo del tipo di ritorno `Unit`

2.3.9 Lambda

Il modo più semplice ed immediato per la definizione di una funzione anonima in Kotlin è quello di utilizzare un'*espressione lambda*; essa possiede la seguente notazione:

{ `arguments -> function body` }

Le lambda di un sola linea di codice in Kotlin non necessitano della parola chiave `return`, poiché restituiscono automaticamente il risultato dell'ultima espressione presente all'interno del loro corpo.

Di seguito vengono elencati alcuni semplici esempi di espressioni lambda:

- { `num1: Int, num2: Int -> num1 + num2` } : espressione lambda che prende in ingresso due argomenti di tipo `Int` e ritorna la somma fra di essi. Il suo tipo è `(Int, Int) -> Int`;
- { `s: String -> print(s)` } : espressione lambda che prende in ingresso un argomento di tipo `String` e lo stampa, non restituendo alcun valore. Il suo tipo è `(String) -> Unit`;
- { `3` } : espressione lambda che non prende in ingresso alcun argomento e ritorna 3. Il suo tipo (*function type*) è `() -> Int`.

La funzione presente a riga 9 del listato 2.35 può ora essere tradotta con una lambda, come segue (listato 2.38):

```
1  val multiplication: (Int, Int) -> Int = { num1: Int, num2: Int -> num1 * num2 }
2  // val multiplication = { num1: Int, num2: Int -> num1 * num2 }   also allowed
3
4  fun sample() {
5      val result = multiplication(4, 5)
6  }
```

Listato 2.38: Esempio di utilizzo di un'espressione lambda in Kotlin

Un'altra caratteristica da non sottovalutare è che le espressioni lambda in Kotlin possono operare anche su elementi diversi dai valori forniti come argomenti in ingresso (possono accedere alla loro cosiddetta *closure*); possono infatti venir utilizzate anche tutte le proprietà e le funzioni inserite nel contesto entro il quale si trova la lambda (listato 2.39):

```
1  val helloText = "Hello"
2  val greeting = { print(helloText) }   // Type: () -> Unit
3
4  fun sample() {
5      greeting()   // Prints 'Hello'
6  }
```

Listato 2.39: Esempio di lambda che utilizza elementi estranei ai valori dei suoi parametri in ingresso

2.3.10 Extensions

Kotlin possiede la capacità di estendere una classe inserendovi nuove funzionalità senza la necessità di ereditare direttamente dalla classe stessa e senza utilizzare pattern appositi come il design pattern Decorator [11]. Questo meccanismo è reso possibile da speciali dichiarazioni chiamate *extensions*; Kotlin ne supporta di due tipi: le *funzioni di estensione* e le *proprietà di estensione*.

Funzioni di estensione

Per comprendere meglio cos'è e come si crea una funzione di estensione in Kotlin, risulta più intuitivo partire da un esempio concreto.

In Android capita spesso di utilizzare i cosiddetti *toast*: brevi messaggi utili per informare l'utente riguardo ad errori o semplicemente per avvisarlo di qualche evento verificatosi (listato 2.40):

```
1  class MainActivity : AppCompatActivity() {
2      override fun onCreate(savedInstanceState: Bundle?) {
3          super.onCreate(savedInstanceState
4          Toast.makeText(this, "Hello everybody!", LENGTH_LONG).show()
5      }
6  }
```

Listato 2.40: Creazione e visualizzazione di un toast in Kotlin su Android

È frequente per i programmatori Android (su Java come su Kotlin) dimenticare almeno una volta l'invocazione del metodo `show()` (fine riga 4), con conseguente mancata visualizzazione del toast e ricerca del problema (a volte non subito evidente).

Kotlin permette in questi casi di creare una funzione di estensione, la quale agisce similmente ad un metodo effettivamente definito all'interno della classe che si decide di estendere. Seguendo questo approccio, è possibile aggiungere la funzione `toast()` come estensione della classe `Context`, richiamarla direttamente all'interno di qualsiasi *activity* (poiché ereditano tutte da `Context`) ed implementarla in modo da prevedere automaticamente la visualizzazione (metodo `show()` discusso nel precedente paragrafo), come da listato 2.41:

```
1  class MainActivity : AppCompatActivity() {
2      override fun onCreate(savedInstanceState: Bundle?) {
3          super.onCreate(savedInstanceState)
4          toast("Hello everybody!")   // Extension function usage
5      }
6
7      private fun Context.toast(text: String) {   // Extension function declaration
8          Toast.makeText(this, text, LENGTH_LONG).show()
9      }
10 }
```

Listato 2.41: Esempio di funzione di estensione in Kotlin

Da notare come dentro il corpo della funzione di estensione possa essere utilizzata la keyword `this` per riferirsi all'oggetto (in questo caso di tipo `Context`) sul quale la funzione viene invocata (riga 8, listato 2.41).

Proprietà di estensione

In maniera analoga a quella discussa per le funzioni, una proprietà di estensione in Kotlin presuppone l'inserimento di nuove proprietà all'interno di una classe senza l'obbligo di ereditare da essa. Un esempio in Android potrebbe essere il seguente (listato 2.42):

```
1  class MainActivity : AppCompatActivity() {
2      private val TextView.trimmedText: String   // Extension property declaration
3          get() = text.toString().trim()
4
5      override fun onCreate(savedInstanceState: Bundle?) {
6          super.onCreate(savedInstanceState)
7          setContentView(R.layout.activity_main)
8          val textView: TextView = findViewById(R.id.textView)
9          Log.i("TAG", textView.trimmedText)   // Extension property usage
10     }
11 }
```

Listato 2.42: Esempio di proprietà di estensione in Kotlin

L'esempio di cui sopra mostra l'aggiunta della proprietà `trimmedText` (la quale in get restituisce il testo privato degli eventuali spazi bianchi di inizio e fine stringa) alla classe `TextView` (righe 2 e 3). Successivamente imposta il layout dell'activity ricavandolo dalle risorse (riga 7), ottiene un riferimento ad un oggetto di tipo TextView (riga 8) e stampa sul *Logcat Monitor* di Android Studio la stringa restituita dal getter della proprietà appena aggiunta (riga 9).

2.3.11 Coroutines

Il modello di threading presente su Android è simile a quello esistente su molti altri framework che cooperano a stretto contatto con l'interfaccia utente (*UI*): si basa su un unico thread (comunemente chiamato *UI Thread* o *Main Thread*) responsabile dell'aggiornamento dell'interfaccia grafica, della cattura degli eventi e di vari altri aspetti che interessano la UI. È quindi comprensibile che questo tipo di architettura possa mostrare dei problemi ogniqualvolta risulti necessaria l'esecuzione di operazioni di lunga durata (cosiddette *long-running operation*), come per esempio query a database, richieste alla rete oppure computazioni particolarmente dispendiose; in questi casi è possibile che l'interfaccia utente vada incontro a blocco forzato e l'applicazione termini segnalando un errore di tipo *Application Not Responding (ANR)*. È proprio in questo scenario che il meccanismo delle *coroutine*[14] può permettere di risolvere il problema in maniera semplice e performante.

Una coroutine in Kotlin consente di svolgere operazioni long-running in maniera asincrona, senza bloccare il thread principale di un'applicazione Android. Può essere pensata come un thread, ma molto più leggero in termini di occupazione di risorse: è possibile infatti creare migliaia di coroutine pagando veramente poco in performance, mentre la stessa situazione con migliaia di thread sarebbe non gestibile su moderni calcolatori.

Per utilizzare una coroutine, è necessario anteporre la parola chiave `suspend` alla dichiarazione di una funzione; questa semplice operazione la rende una co-

[14]Il meccanismo delle coroutine è ancora in fase sperimentale dalla versione 1.1 di Kotlin, il che significa che l'API continuerà ad evolvere nel tempo.

siddetta *funzione di sospensione* (*suspending function*), poiché d'ora in poi la sua chiamata determinerà la sospensione di una coroutine. Una funzione di sospensione può ricevere in ingresso parametri e restituire valori nella stessa maniera utilizzata da una normale funzione, ma con una limitazione: può essere richiamata solo da coroutine o da altre suspending function.

Per la creazione di coroutine è possibile utilizzare due builder appositi:

- `launch {...}` : crea coroutine senza bloccare il thread corrente e restituisce un suo riferimento come oggetto della classe `Job` (il quale può poi essere utilizzato per cancellare la coroutine stessa);
- `async {...}` : crea coroutine e restituisce il loro futuro risultato sotto forma di oggetto della classe `Deferred`.

Un esempio può essere il seguente (listato 2.43):

```
1 async {
2     ...
3     val result = computation.await()
4     ...
5 }
```

Listato 2.43: Esempio di utilizzo di coroutine in Kotlin

Il metodo `await()` (riga 3) può essere una funzione di sospensione (quindi richiamabile all'interno del blocco `async {...}`, riga 1) che sospende la coroutine finché la computazione non è terminata ed è stato restituito il risultato.

2.4 Novità di Kotlin 1.2

In questa sezione verranno elencate e discusse brevemente le principali caratteristiche e funzionalità aggiunte con l'uscita della versione 1.2 di Kotlin.

In Kotlin 1.1 è stata resa possibile la compilazione del codice Kotlin in JavaScript mentre ora, con la versione 1.2, è possibile anche riusare il codice tra JVM e JavaScript. È quindi consentito scrivere la logica dell'applicazione Kotlin una sola volta, per poi riusarla all'interno di molteplici parti del sistema. Sono inoltre state migliorate alcune librerie al fine di incrementare il riuso di codice.

La filosofia a capo di questa versione viene perciò probabilmente ben condensata con la frase 'Sharing Code between Platforms'.

2.4.1 Progetti multipiattaforma

Un progetto multipiattaforma permette di creare differenti livelli della stessa applicazione (come per esempio backend, frontend e app per Android) dalla stessa base di codice. Come evidenziato in figura 2.5, un progetto di questo tipo contiene

sia moduli comuni (*common modules*), che contengono codice indipendente dalla piattaforma, sia moduli specifici (*platform-specific modules*), i quali possiedono invece codice per una data architettura (JVM o JS) e che utilizzano librerie per precise piattaforme.

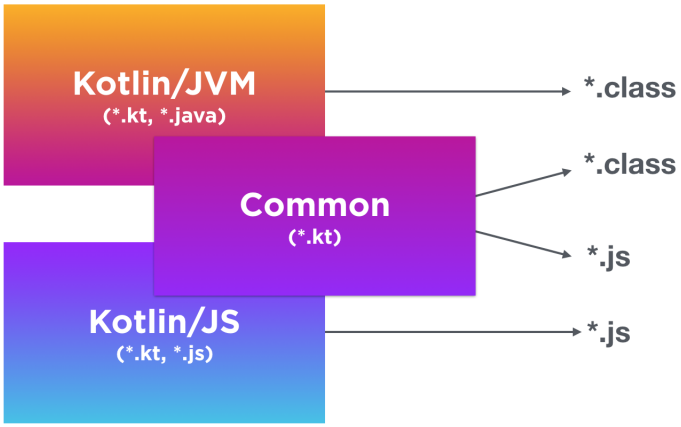


Figura 2.5: Tipi di moduli presenti in un progetto multipiattaforma in Kotlin (v1.2)

Va ricordato che i progetti multipiattaforma sono attualmente in fase sperimentale; questo significa che possono già essere utilizzati, ma che potrebbe essere necessario modificare il progetto nelle versioni successive di Kotlin.

2.4.2 Prestazioni in compilazione

Durante lo sviluppo della versione 1.2 di Kotlin, molta importanza è stata data alla velocità del processo di compilazione; ed evidentemente gli sforzi dei progettisti e degli sviluppatori hanno portato i loro frutti: questa versione è oltre il 25% più veloce in compilazione rispetto a Kotlin 1.1.

La figura 2.6 sottostante mostra le differenze dei tempi di compilazione fra due progetti di grandi dimensioni scritti rispettivamente in Kotlin 1.1 e 1.2:

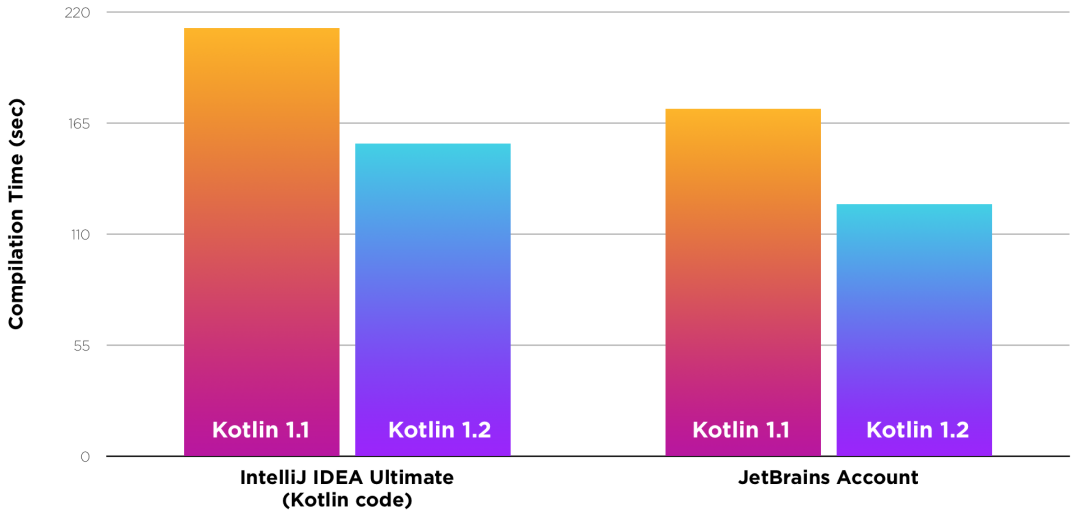


Figura 2.6: Differenze di prestazione in compilazione tra Kotlin 1.1 e 1.2

2.4.3 Altri miglioramenti del linguaggio e delle librerie

Altri miglioramenti sono stati apportati ad alcune librerie:

- **kotlin.test** : ora permette di scrivere codice una sola volta ed eseguirlo su sia su sistemi JVM che su JS;
- **kotlinx.html** : ora supporta il *rendering isomorfo*, utilizzando lo stesso codice per il rendering HTML sia su backend che su frontend;
- **kotlinx.serialization** : consente di utilizzare oggetti Kotlin tra diversi livelli di un'applicazione, utilizzando JSON o ProtoBuf come formati di serializzazione.

Sono inoltre stati effettuati piccoli miglioramenti sul linguaggio e sulla libreria standard:

- una sintassi più concisa per inviare molteplici argomenti a un'annotazione (*array literals*);
- il supporto per il modificatore `lateinit`, utilizzato su proprietà e variabili locali;
- cast più 'intelligenti' e miglioramenti generali della type inference;
- compatibilità della libreria standard con le limitazioni sui package divisi (*split package*) introdotti in Java 9;
- nuovo package *kotlin.math* inserito nella libreria standard;

- nuove funzioni di libreria standard per lavorare con le sequenze e le collezioni, incluse quelle per dividere una collezione (o sequenza) in gruppi (potenzialmente sovrapposti) con lunghezza fissa.

Capitolo 3

RxKotlin su Android

RxKotlin è una libreria che implementa i principali concetti della programmazione reattiva per il linguaggio Kotlin. È inoltre in buona parte influenzata dal paradigma di programmazione funzionale, fornendo difatti strumenti per la composizione in cascata degli operatori (funzioni) e per la rimozione degli effetti collaterali.

RxKotlin possiede quindi il modello produttore/consumatore (osservabile/osservatore) tipico del mondo reattivo, introducendo tuttavia anche funzioni per la composizione, trasformazione, filtraggio e manipolazione dei flussi di dati osservabili.

3.1 Perché RxKotlin

Nella presente tesi si è scelto di descrivere RxKotlin per diversi motivi: innanzitutto perché è un potente strumento in grado di fondere insieme sia le utili caratteristiche innovative del moderno linguaggio Kotlin, sia le idee e i concetti già discussi del paradigma di programmazione reattiva. Inoltre, lo si è scelto anche poiché Kotlin è un linguaggio di programmazione ufficialmente supportato per lo sviluppo di applicazioni Android; risulta dunque la scelta più naturale, dato che questa tesi tratta l'approccio della Reactive Programming al mondo Android.

3.1.1 Differenze e affinità con RxJava

RxKotlin è una libreria che aggiunge utili funzioni ed estende nella pratica RxJava [18]. La prima può quindi essere vista come un 'wrapper' scritto in Kotlin della seconda, integrando quindi tutte le funzionalità linguistiche e i miglioramenti sintattici che hanno reso famoso questo moderno linguaggio rispetto al predecessore. In ultima analisi: RxKotlin si propone di ereditare e raccogliere tutte le caratteristiche di RxJava in una sola leggera libreria migliorata e di standardizzare le convenzioni per l'utilizzo di RxJava con Kotlin.

Le due librerie risultano equivalenti e interoperabili dal punto di vista delle funzionalità, sebbene RxKotlin risulti migliore per quanto riguarda la leggibilità del codice, la concisione e i servizi linguistici offerti al programmatore [22].

Esiste tuttora un supporto di RxKotlin per entrambe le versioni di RxJava (RxJava 1 e 2); nel prosieguo della tesi si descriverà RxKotlin 2, la versione attualmente più recente.

3.2 Fasi preliminari

Di seguito sono elencati i passi necessari per importare la libreria RxKotlin 2 in progetti *gradle* [24] sull'IDE Android Studio al fine di poter utilizzare le feature messe a disposizione.

3.2.1 Importare RxKotlin 2 su Android Studio

Ad oggi esiste una seconda versione di RxKotlin, per importarla su un progetto in Android Studio è sufficiente aggiungere le seguenti due linee nelle `dependencies` del file `build.gradle` dell'applicazione:

```
implementation 'io.reactivex.rxjava2:rxandroid:2.x.y'
implementation 'io.reactivex.rxjava2:rxkotlin:2.w.z'
```

dove `x.y` e `w.z` vanno sostituiti con i numeri delle versioni che si intendono importare; attualmente le ultime versioni sono, rispettivamente per RxAndroid e RxKotlin, la `2.1.0` e la `2.2.0`.

È bene precisare che non è necessario importare RxAndroid per programmare in RxKotlin, tuttavia è nella pratica quasi un obbligo nel caso si programmi su smartphone, poiché la libreria (estensione reattiva per Android) fornisce diversi *scheduler* per eseguire codice su thread separati e sul Main Thread di Android. In questo modo si possono facilmente eseguire compiti su un thread in background per poi mostrare il risultato tramite interfaccia grafica (utilizzando l'UI Thread). Questo consente per esempio di sostituire l'implementazione di un *AsyncTask*, risparmiando diverse linee di codice e rendendo il listato generalmente più chiaro e leggibile.

Successivamente è necessario (se non è già impostato) configurare Android Studio al fine di poter utilizzare le feature di Java 8; RxKotlin utilizza infatti un approccio di programmazione orientato al funzionale, dunque risulta utile (e consigliato) utilizzare alle volte lambda expression e method reference (introdotti dalla versione 8 di Java). Per assolvere al compito è sufficiente aggiungere due linee nel file *build.gradle* dell'app, come segue:

```
1 android {
2   // ...
3   compileOptions {
4     sourceCompatibility JavaVersion.VERSION_1_8
5     targetCompatibility JavaVersion.VERSION_1_8
6   }
7 }
```

3.3 Concetti di base

All'interno di questa sezione verranno affrontati i concetti alla base del framework RxKotlin, soffermandosi principalmente sulla descrizione degli *observable*, degli *observer* e dei *subject*.

3.3.1 Entità osservabili

Come già accennato precedentemente, RxKotlin estende a livello logico il pattern Observer [11]. Questa caratteristica fornisce alla libreria due fondamentali classi di entità: quelle osservabili e quelle osservatrici. Le prime, chiamate spesso *observable*[1], rappresentano sorgenti di dati o eventi e vengono reificate in oggetti di RxKotlin (presto verranno descritti uno per volta).

Questa tipologia di entità è stata già precedentemente descritta a livello teorico nella sottosezione 1.2.6 del Capitolo 1. Di seguito verrà però presentata nell'ambito della programmazione in RxKotlin.

Tipologie e conversioni

Di seguito vengono esposte le varie tipologie di *entità osservabili* presenti in RxKotlin 2, le loro caratteristiche di base e i metodi di conversione tra l'una e l'altra tipologia[2]. Ognuna di esse è rappresentata nella prossima tabella da una specifica classe della libreria:

[1] Così vengono chiamate le entità osservabili in RxKotlin 2, da non confondere con la classe `Observable` (da notare l'iniziale maiuscola e il diverso font), presente all'interno del framework.

[2] Attenzione: le tipologie di observable (quindi le classi) cambiano leggermente da RxKotlin a RxKotlin 2, nella presente tesi viene descritta la più recente.

Tipologia	**Descrizione**
`Flowable<T>`	Emette 0 oppure n (numero qualsiasi) elementi e termina con un evento di successo o di errore. Supporta backpressure[3], tecnica che permette di risolvere il problema che si presenta quando i dati/eventi vengono emessi ad una velocità superiore rispetto a quella di elaborazione da parte dell'osservatore.
`Observable<T>`	Emette 0 oppure n elementi e termina con un evento di successo o di errore. Non supporta backpressure.
`Single<T>`	Emette un singolo elemento oppure un evento di errore. Non supporta backpressure.
`Maybe<T>`	Emette 0 elementi, un singolo elemento oppure un evento di errore. Non supporta backpressure.
`Completable`	Non emette elementi, termina con un evento di successo o di errore. Non supporta backpressure.

Come si evince dalla precedente tabella, `Flowable` e `Observable` rappresentano stream di dati che possiedono lo stesso comportamento, l'unica differenza fra i due è che il primo supporta backpressure mentre il secondo ne è sprovvisto. Anche `Single`, `Maybe` e `Completable` sono sprovvisti di tecnica backpressure, si capirà più avanti che questo è ovvio (e necessario) poiché essi emettono al più un solo elemento.

Nelle prossime tabelle vengono invece esposti i nomi dei metodi da chiamare per convertire un oggetto da una data tipologia di observable ad un'altra. Da tener presente che a volte ci sono più metodi disponibili oppure che alcuni richiedono una certa logica per essere attuati correttamente:

[3]Argomento di cui si parlerà in maniera più approfondita successivamente.

Da \ A	**Flowable**	**Observable**	**Single**
Flowable		toObservable	first, firstOrError, single, singleOrError, last, lastOrError[4]
Observable	toFlowable[5]		first, firstOrError, single, singleOrError, last, lastOrError[4]
Single	toFlowable[6]	toObservable	
Maybe	toFlowable[6]	toObservable	toSingle
Completable	toFlowable	toObservable	toSingle

Da \ A	**Maybe**	**Completable**
Flowable	firstElement, singleElement, lastElement	ignoreElements
Observable	firstElement, singleElement, lastElement	ignoreElements
Single	toMaybe	toCompletable
Maybe		ignoreElement
Completable	toMaybe	

[4]La presenza di più metodi è dovuta al fatto che durante la conversione da una sorgente multivalore ad una monovalore bisogna decidere quale elemento considerare come risultato della conversione.

[5]Convertire un `Observable` (privo di backpressure) in un `Flowable` necessita di prendere una decisione: cosa fare con il flusso della sorgente `Observable`. Ci sono diverse strategie disponibili (come per esempio il buffering, la perdita di elementi oppure il mantenimento dell'ultimo valore) utilizzabili tramite le costanti definite nella enum `BackpressureStrategy` oppure tramite gli operatori standard di `Flowable` come `onBackpressureBuffer`, `onBackpressureDrop` e `onBackpressureLatest`, i quali permettono anche una personalizzazione ulteriore del comportamento di backpressure.

[6]Quando c'è solo (al massimo) un elemento emesso dalla sorgente, non vi è alcun problema di backpressure in quanto il valore può essere sempre memorizzato fino a quando il downstream non è pronto per riceverlo.

Creazione di observable

In questa sottosezione viene spiegato come avviene la creazione di oggetti observable, ricordando che con il termine *observable* ci si riferisce in generale alle entità osservabili di RxKotlin 2 (e non ad una loro specifica tipologia). Si incomincia per semplicità dalla classe `Observable`, listato 3.1:

```
1  val observable: Observable<Int> = Observable.create { e ->
2      for (i in 0..3) {
3          e.onNext(i)
4          try {
5              Thread.sleep(1000)
6          } catch (exception: InterruptedException) {
7              e.onError(exception)
8          }
9      }
10     e.onComplete()
11 }
```

Listato 3.1: Creazione di un `Observable` in RxKotlin 2

Viene creato un oggetto `Observable` che emette (`onNext()` riga 3) quattro interi da 0 a 3, uno dopo l'altro ad intervalli di un secondo (`Thread.sleep(1000)` riga 5). Tipicamente gli oggetti observable non iniziano a trasmettere dati finché un'entità osservatrice non vi si sottoscrive. In questo esempio non avvengono sottoscrizioni, viene solamente creato un oggetto `Observable` e descritto il comportamento che attuerà una volta ricevuta una sottoscrizione da parte di un oggetto osservatore. Per questi motivi lo snippet di codice non esegue ancora nessun compito, tuttavia è utile inizialmente per comprendere alcuni meccanismi di base. È particolarmente importante sottolineare anche che a riga 5 il metodo `sleep()` viene eseguito dal Main Thread, bloccando l'interfaccia dell'app Android per un secondo ad ogni iterazione del ciclo *for*, di certo non un'ottima scelta; presto verrà spiegato come risolvere il problema eseguendo i task long-running in un thread separato. Infine è bene precisare come questo sia solo un esempio di base e mostri volutamente tutto ciò che succede passo passo, sebbene generalmente non venga quasi mai utilizzato tanto codice per la creazione di un `Observable` in reali progetti con RxKotlin 2; in seguito verranno descritti semplici tecniche per diminuire drasticamente le linee di codice necessarie.

Come è possibile intuire nel precedente listato, un oggetto `Observable` (in generale ogni tipo di observable) è molto simile ad un oggetto che implementa l'interfaccia `Iterable` di Java: data una sequenza, la itera ed emette ciascun elemento in ordine. Ogni volta che un `Observable` emette un elemento, notifica le sue entità osservatrici sottoscritte utilizzando il metodo `onNext(elem)` (riga 3). Una volta che ha trasmesso tutti i suoi valori, segnala la terminazione con successo chiamando il metodo `onComplete()` (riga 10). Se d'altra parte, durante la trasmissione di elementi, l'`Observable` si imbatte in un errore, termina con insuccesso notificando gli osservatori tramite il metodo `onError(throwable)` (riga 7). Questo compor-

tamento può essere formalizzato con la seguente espressione regolare:

```
(onNext)* (onError | onComplete)
```

Ora che si sono compresi i meccanismi di creazione di un oggetto `Observable`, non sarà complicato capire anche come avviene la creazione di un altro tipo observable, chiamato `Flowable` (per altro presente solo nella seconda versione di RxKotlin ed RxJava, listato 3.2):

```
1  val flowable: Flowable<String> = Flowable.create({ e ->
2      for (i in 0..3) {
3          e.onNext(i.toString())
4          try {
5              Thread.sleep(1000)
6          } catch (exception: InterruptedException) {
7              e.onError(exception)
8          }
9      }
10     e.onComplete()
11 }, BackpressureStrategy.DROP)
```

Listato 3.2: Creazione di un `Flowable` in RxKotlin 2

Il meccanismo è praticamente identico a quello visto in precedenza (qui però si è deciso di emettere stringhe invece che interi): viene creato un oggetto `Flowable` che, una volta sottoscritto da un osservatore, inizierà ad emettere in sequenza (riga 3) le stringhe da '0' a '3' intervallate da un secondo di pausa (riga 5). Nel caso avvenga un errore di interruzione verrà segnalato (riga 7), se invece tutto andrà a buon fine verrà notificato con il solito metodo `onComplete()` (riga 10). La regular expression che sintetizza questo comportamento è la stessa indicata per l' `Observable`.

L'unica differenza sostanziale tra un `Observable` e un `Flowable`, come già ripetutamente specificato, è che quest'ultimo supporta *backpressure*; il "prezzo da pagare" risiede perciò nel dover obbligatoriamente specificare come secondo parametro del metodo `create()` la strategia di backpressure che si intende adottare (riga 11). In questo caso è stata indicata tramite un'enumerazione della classe `BackpressureStrategy`.

La procedura per creare un observable di tipo `Single` è molto simile alle precedenti, con la differenza che questa tipologia emette un solo elemento oppure un messaggio di errore (listato 3.3):

```
1  val single: Single<Float> = Single.create { e ->
2      try {
3          Thread.sleep(2000)
4      } catch (exception: InterruptedException) {
5          e.onError(exception)
6      }
7      e.onSuccess(3.5f)
```

```
8 }
```

Listato 3.3: Creazione di un `Single` in RxKotlin 2

Nell'esempio sopra, al momento della sottoscrizione da parte di un'entità osservatrice, l'observable viene messo in pausa per 2 secondi (riga 3) e poi viene emesso un valore float (`onSuccess()`, riga 7). Il comportamento di un `Single` viene formalizzato quindi con la seguente espressione regolare:

`(onSuccess | onError)`

Come si evince, il medoto `onSuccess()` condensa in una sola chiamata i concetti precedentemente visti per `onNext()` e `onComplete()`. Un oggetto `Single` può infatti emettere solo un elemento, perciò nel momento dell'emissione è certo che abbia anche completato con successo il suo compito. Più chiamate ripetute al metodo `onSuccess()` non producono alcun effetto.

Per creare un oggetto di tipo `Maybe` si può seguire la stessa identica procedura descritta per `Single`, con la differenza che `Maybe` può emettere *al più* un valore, il ché gli permette anche di non emettere alcun elemento, terminando con successo (listato 3.4):

```
1 val maybe: Maybe<Double> = Maybe.create { e ->
2     try {
3         Thread.sleep(2000)
4     } catch (exception: InterruptedException) {
5         e.onError(exception)
6     }
7     // e.onSuccess(10.12E24);
8     e.onComplete()
9 }
```

Listato 3.4: Creazione di un `Maybe` in RxKotlin 2

Perciò per `Maybe` torna utile il metodo di terminazione con successo `onComplete()` (riga 8), da chiamare solo nel caso non ci sia un elemento da emettere. In caso contrario si dovrà utilizzare soltanto `onSuccess(elem)` (commentato a riga 7). Perciò l'espressione regolare che formalizza il comportamento mutualmente esclusivo di un oggetto observable ti tipo `Maybe` è la seguente:

`(onSuccess | onError | onComplete)`

Infine viene descritto l'observable di tipo `Completable` che, si ricorda, non emette mai alcun valore (da notare nel prossimo listato 3.5 la classe non generica e la mancanza dei metodi `onNext()` o `onSuccess()`). Il suo compito è quello di segnalare un messaggio di errore oppure di completamento con successo, per questo viene spesso utilizzato per compiere test o quando si necessita di creare velocemente oggetti observabili. L'espressione regolare che rappresenta il suo comportamento

in fase di sottoscrizione è la seguente:

`(onError | onComplete)`

```
1  val completable = Completable.create { e ->
2      try {
3          Thread.sleep(1500)
4      } catch (exception: InterruptedException) {
5          e.onError(exception)
6      }
7      e.onComplete()
8  }
```

Listato 3.5: Creazione di un `Completable` in RxKotlin 2

Blocca per 1.5 secondi il Main Thread (riga 3) per poi segnalare il completamento (riga 7). Se durante lo *sleep* si verifica una `InterruptedException` non viene chiamato `onComplete()`, bensì `onError()`.

Metodi di creazione agili

RxKotlin 2 mette a disposizione anche approcci molto più veloci per la creazione di observable, tramite l'uso dei seguenti metodi[7]:

- `Observable.just()`

  Questo metodo converte ogni oggetto passato come argomento in un elemento da poter emettere. Il risultante observable, se sottoscritto ad un'observer, emetterà in sequenza tutti gli oggetti inseriti e poi, in caso di assenza di errori, segnalerà il suo completamento:

```
1  val obs = Observable.just("First", "Second", "Third", "Fourth", "Fifth")
```

  Nota: il metodo `just()` accetta al massimo 10 parametri in ingresso.

- `Observable.from()`

  Comprende una famiglia di metodi tra cui `fromArray()`, `fromIterable()`, `fromCallable()`, `fromFuture()` e `fromPublisher()`, i quali consentono di convertire una collezione di oggetti in uno stream osservabile; di seguito due esempi:

```
1  val obs = Observable.fromArray(1, 2, 3, 4)
```

[7]Nei prossimi listati viene mostrato `Observable` per semplicità e chiarezza, ma la la stessa procedura si può attuare per tutte le altre tipologie già descritte di observable.

```
1 val sauces = Arrays.asList("ketchup", "mayo")
2 val obs = Observable.fromIterable<String>(sauces)
```

- `Observable.range()`

  Comprende i metodi `Observable.range()` e `Observable.rangeLong()`; necessitano di due parametri (valore iniziale e valore finale) e consentono di creare un observable che, una volta sottoscritto, emetterà un intervallo ordinato rispettivamente di interi e di long:

```
1 val obs = Observable.range(10, 50)
```

```
1 val obs = Observable.rangeLong(10000L, 80000L)
```

- `Observable.interval()`

  Tramite questo metodo si crea un observable che, una volta sottoscritto, emetterà una sequenza crescente infinita di interi, intervallati fra di loro dalla quantità di tempo indicata nell'argomento (nell'esempio 2 secondi):

```
1 val obs = Observable.interval(2, TimeUnit.SECONDS)
```

- `Observable.empty()`

  Questo metodo crea un observable che, una volta sottoscritto, non emetterà elementi ma terminerà normalmente oppure con un errore; può essere utile quando si ha bisogno di creare velocemente un observable a scopo di test:

```
1 val obs = Observable.empty<String>()
```

Cold observable

I *cold observable* sono una categoria di observable che inizia ad emettere dati ed eventi nel momento preciso della sottoscrizione da parte di un observer. Inoltre, ripropone la sequenza di emissioni dall'inizio a ciascun nuovo iscritto. Per fare un esempio: tutti gli observable creati precedentemente tramite l'utilizzo dei metodi agili sono cold observable; indipendentemente da quando vengono creati, riemettono i valori per ogni sottoscrizione.

I cold observable non presentano tuttavia necessariamente la stessa identica sequenza a ciascun observer. Se, ad esempio, un cold observable si connette a un

database ed emette i risultati di alcune query, i valori effettivi dipenderanno dallo stato del database al momento della sottoscrizione.

Nel prossimo listato 3.6 viene mostrato un esempio di utilizzo di un cold observable e relativo output sottostante:

```
1  val coldObs = Observable.interval(200, TimeUnit.MILLISECONDS)
2
3  coldObs.subscribe { i -> Log.i("TAG", "First: $i") }
4  Thread.sleep(500)
5  coldObs.subscribe { i -> Log.i("TAG", "Second: $i") }
6
7  /*
8  OUTPUT:
9  First: 0
10 First: 1
11 First: 2
12 Second: 0
13 First: 3
14 Second: 1
15 First: 4
16 Second: 2
17 ...
18 */
```

Listato 3.6: Esempio di utilizzo di un cold observable

I due observer (uno a riga 3, l'altro a riga 5) non ricevono gli stessi valori nello stesso tempo, anche se entrambi sono sottoscritti allo stesso observable. "Vedono" la stessa sequenza di elementi, tranne per il fatto che ognuno di loro la vede come iniziata nel preciso momento in cui si è sottoscritto (i momenti di sottoscrizione differiscono fra loro di 500 millisecondi, riga 4); la variabile `coldObs` (riga 1) rappresenta quindi un tipico esempio di cold observable.

Hot observable

Gli *hot observable* sono un'altra categoria di observable che emette valori indipendentemente dalle singole sottoscrizioni. Questo tipo di entità osservabili possiede la propria sequenza temporale ed emette elementi in maniera autonoma, a prescindere dall'ascolto o meno da parte di potenziali observer. Un esempio di hot observable è per esempio quello che emette eventi provenienti dal tocco dell'utente sullo schermo dello smartphone in un'applicazione Android. L'utente, infatti, genera eventi di tocco indipendentemente dal fatto che esista un oggetto che li ascolti.

Inoltre, al momento della sottoscrizione, gli hot observable non ricevono la sequenza degli elementi dall'inizio, ma captano gli eventi attuali mentre accadono: non si riceve e non si desidera ricevere difatti un riepilogo di tutti gli elementi emessi dalla creazione dell'observable. Se si annulla la sottoscrizione, l'entità osservabile continua ad emettere dati anche in assenza di osservatori; se ci si sotto-

scrive nuovamente, si iniziano a ricevere gli eventi correnti senza poter recuperare quelli eventualmente persi.

Nel prossimo listato 3.7 viene mostrato un esempio di utilizzo di un hot observable in Android:

```
1  class MainActivity : AppCompatActivity() {
2      override fun onCreate(savedInstanceState: Bundle?) {
3          super.onCreate(savedInstanceState)
4          setContentView(R.layout.activity_main)
5
6          val button = findViewById<Button>(R.id.button)
7          val textView = findViewById<TextView>(R.id.textView)
8          textView.text = 0.toString()    // Initial value
9
10         val observable = Observable.create<Int> { e ->
11             button.setOnClickListener {
12                 e.onNext(textView.text.toString().toInt() + 1)
13             }
14
15             e.setCancellable {
16                 button.setOnClickListener(null)
17                 e.onComplete()
18             }
19         }
20
21         observable.subscribe(object : Observer<Int> {
22             override fun onNext(number: Int) {
23                 textView.text = number.toString()
24             }
25
26             override fun onSubscribe(d: Disposable) { }
27             override fun onError(e: Throwable) { }
28             override fun onComplete() { }
29         })
30     }
31 }
```

Listato 3.7: Esempio di utilizzo di un hot observable in Android

Nell'esempio, l'observable emette un numero intero incrementale (riga 12) ogniqualvolta l'utente clicca sul bottone presente nel layout della main activity, mentre l'observer sottoscritto aggiorna una textview di conseguenza (riga 23).

3.3.2 Entità osservatrici

Le entità osservatrici, chiamate spesso *observer*[8], rappresentano recettori che possono sottoscriversi ad entità osservabili per ricevere gli elementi emessi.

Le entità osservatrici sono state già precedentemente descritte a livello teorico nella sottosezione 1.2.6 del Capitolo 1. Di seguito verranno però presentate nell'ambito della programmazione in RxKotlin.

[8]Così vengono chiamate le entità osservatrici in RxKotlin 2, da non confondere con la classe `Observer` (da notare l'iniziale maiuscola e il diverso font), presente all'interno del framework.

Tipologie

Esistono vari tipi di observer, ognuno specifico per una data tipologia di observable:

Tipologia *observable*	**Tipologia *observer***
`Flowable<T>`	`Subscriber<T>`
`Observable<T>`	`Observer<T>`
`Single<T>`	`SingleObserver<T>`
`Maybe<T>`	`MaybeObserver<T>`
`Completable`	`CompletableObserver`

Creazione di observer

In questa sottosezione viene spiegato come avviene la creazione di oggetti observer. Ci si limita a presentare il metodo di creazione di un oggetto della classe `Observer`, tenendo conto che gli altri observer vengono creati similmente (facendo l'override dei metodi chiamati dal corrispondente observable).

Nota: nel seguente listato 3.8 si supporrà che l'oggetto di nome *observable* (riga 19) sia già stato creato con le tecniche esposte in precedenza.

```
1  val observer = object : Observer<Int> { // anonymous class, inherits from Observer
2      override fun onSubscribe(d: Disposable) {
3          Log.i("TAG", "onSubscribe")
4      }
5
6      override fun onNext(value: Int) {
7          Log.i("TAG", "onNext: $value")
8      }
9
10     override fun onError(exc: Throwable) {
11         Log.e("TAG", "onError: $exc")
12     }
13
14     override fun onComplete() {
15         Log.i("TAG", "onComplete: All Done!")
16     }
17 }
18
19 observable.subscribe(observer)
```

Listato 3.8: Creazione di un `Observer` in RxKotlin 2

L'esempio mostra un oggetto di tipo `Observer` che stampa stringhe su *Logcat Monitor* in base al metodo chiamato dal suo oggetto di tipo `Observable`, al quale si sottoscrive a riga 19.

Subito dopo l'atto della sottoscrizione, l'entità osservabile chiama il metodo `onSubscribe()` (riga 2) al quale passa come unico argomento un oggetto `Disposable`; tramite tale oggetto sarà possibile per l'osservatore revocare la sottoscrizione in ogni momento, nella maniera seguente (listato 3.9):

```
1 d.dispose()
```

Listato 3.9: Revoca della sottoscrizione da parte di un observer in RxKotlin 2

Dal momento di tale revoca in poi, l'observer non riceverà più elementi dall'observable al quale era 'agganciato', a meno di una nuova richiesta di sottoscrizione.

3.3.3 Subjects

In RxKotlin 2, i cosiddetti *subject*, generati dalla classe `Subject`, sono oggetti capaci contemporaneamente sia di emettere elementi (come un observable), sia di osservarli (come un observer). Essi possiedono quindi l'abilità di sottoscriversi ad una o più entità osservabili, ma anche di emettere dati/eventi verso i propri sottoscrittori (osservatori). Questa "doppia" capacità è resa possibile grazie al fatto che la classe `Subject` estende la classe `Observable` ed implementa quella `Observer`.

I subject sono spesso utilizzati quando è necessario lavorare con eventi in multicasting (cioè quando bisogna inviare dati a più destinazioni nello stesso tempo) e per convertire cold observable in hot observable.

`Subject` è la classe da cui estendono `AsyncSubject`, `BehaviorSubject`, `PublishSubject`, `ReplaySubject`, `UnicastSubject`, `CompletableSubject`, `SingleSubject` e `MaybeSubject`. Nel prossimo listato 3.10 viene mostrato un semplice utilizzo di un subject di tipo `BehaviorSubject`:

```
1 val subject = BehaviorSubject.create<String>()
2
3 val observable = Observable.just("cold", "warm", "hot")
4
5 // Subject used as observer
6 observable.subscribe(subject)
7
8 // Subject used as observable
9 subject.subscribe { s -> Log.i("TAG", s) }
```

Listato 3.10: Esempio di utilizzo di un subject in RxKotlin 2

3.4 Caratteristiche avanzate

In questa sezione vengono presentate alcune delle caratteristiche di RxKotlin 2 che non fanno parte delle nozioni di base. Vengono dunque descritte le varie tipologie di *scheduler*, l'approccio al multithreading reso possibile con l'aggiunta della libreria RxAndroid 2, il fenomeno della backpressure (con le varie strategie di attuazione) e gli operatori per la manipolazione dei flussi di dati.

3.4.1 Schedulers

Gli scheduler in RxKotlin 2 sono responsabili dell'esecuzione delle operazioni di observable su thread differenti; aiutano dunque la divisione del carico di lavoro su diversi flussi d'esecuzione.

Di seguito vengono elencati i differenti tipi di scheduler e viene spiegato in quali situazioni utilizzarli:

- `Schedulers.io()`

  Questa è una delle tipologie più comuni di scheduler che vengono normalmente impiegati. È adatta per compiti prevalentemente *IO-bound*, come richieste di rete o accessi a file system. Possiede un *pool* di thread espandibile in base alle esigenze. Può essere utilizzato nel modo seguente:

```
1 observable.subscribeOn(Schedulers.io())
```

- `Schedulers.computation()`

  Questo scheduler è utile per piccole operazioni prevalentemente *CPU-bound*, sebbene presenti limitazioni: può infatti utilizzare un numero massimo di thread pari al numero di core presenti sul sistema in uso. Se si hanno a disposizione 2 core su uno smartphone, per esempio, questo scheduler possiederà perciò un pool composto da soli 2 thread. Ciò significa che quando questi thread saranno occupati, il processo dovrà attendere che tornino ad essere disponibili. Risulta dunque adatto per l'esecuzione rapida di piccoli calcoli. Può essere utilizzato nel modo seguente:

```
1 observable.subscribeOn(Schedulers.computation())
```

- `Schedulers.newThread()`

  Come suggerisce il nome, genera un nuovo thread per ogni nuova unità di lavoro. Può essere perciò usato per eseguire operazioni in background che richiedono molto tempo (long-running). Siccome però genera un thread ogni

volta che è necessario, risulta importante tener sotto controllo il loro numero, poiché la creazione di thread è un'operazione costosa e può avere effetti negativi, sopratutto in ambienti mobile. Può essere utilizzato nel modo seguente:

```
observable.subscribeOn(Schedulers.newThread())
```

- `Schedulers.single()`

  Questo scheduler è piuttosto semplice da utilizzare in quanto è supportato solo da un singolo thread. Quindi non importa quante unità di lavoro esistano in un dato momento: esso funzionerà sempre soltanto su un thread. Può essere utilizzato nel modo seguente:

```
observable.subscribeOn(Schedulers.single())
```

- `Schedulers.trampoline()`

  Questo scheduler esegue sul thread corrente; quindi se un frammento di codice è in esecuzione sul Main Thread, questo scheduler aggiungerà il blocco alla coda del thread principale. Gli scheduler appartenenti a questa tipologia sono utili quando si possiedono più observable ed è necessario eseguirli in ordine. Di seguito un esempio di utilizzo (listato 3.11):

```
Observable.just(0, 1, 2, 3)
          .subscribeOn(Schedulers.trampoline())
          .subscribe { Log.i("TAG", "Number: $it") }

Observable.just(4, 5, 6)
          .subscribeOn(Schedulers.trampoline())
          .subscribe { Log.i("TAG", "Number: $it") }
/*
OUTPUT:
Number: 0
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
...
*/
```

Listato 3.11: Utilizzo di uno scheduler di tipo *trampoline* in RxKotlin 2

- `AndroidSchedulers.mainThread()`

  Questo scheduler viene fornito dalla libreria RxAndroid è sarà descritto maggiormente all'interno della prossima sottosezione. É usato per riportare l'esecuzione da un thread di background a quello principale di Android, in modo che possa venir modificata l'interfaccia utente. Può essere utilizzato nel modo seguente:

```
1 observable.observeOn(AndroidSchedulers.mainThread())
```

3.4.2 Multithreading con RxAndroid

Come già precisato, alcuni esempi precedenti, seppur corretti, presentano un'inefficienza: il metodo `Thread.sleep()` viene infatti eseguito dal Main Thread, bloccando l'interfaccia dell'applicazione Android e diminuendo perciò la reattività della stessa. Per risolvere con facilità ed efficacia questo tipo di problema è possibile utilizzare la libreria denominata *RxAndroid*.

RxAndroid mette a disposizione uno scheduler per eseguire porzioni di codice direttamente sull'UI Thread. Grazie a questo scheduler è possibile definire un observable che svolga compiti long-running su un thread di background, per poi emettere i risultati della sua computazione direttamente sull'interfaccia grafica (utilizzando il Main Thread). In questo modo si rende l'app reattiva, sempre responsiva agli eventuali input provenienti dall'interno e dall'esterno. Il meccanismo appena descritto consente di rimpiazzare per esempio un complesso e a volte poco comprensibile *AsyncTask* in poche eleganti righe di codice molto più leggibili (listato 3.12):

```
1 Observable.just("Alfa", "Beta", "Gamma", "Delta")
2             .subscribeOn(Schedulers.newThread())
3             .observeOn(AndroidSchedulers.mainThread())
4             .subscribe(observer)
```

Listato 3.12: Esempio di utilizzo di RxAndroid

Il metodo `subscribeOn()` (riga 2) specifica quale tipo di scheduler debba prendersi carico della computazione (creazione dell'observable, in questo specifico caso). Si ricorda che `Schedulers.newThread()` è uno scheduler che crea un nuovo thread per ogni unità di lavoro, dunque particolarmente utile per l'esecuzione di operazioni long-running in background. Il metodo `observeOn()` (riga 3), invece, specifica quale scheduler ha il compito di mostrare il risultato del lavoro svolto in background. `AndroidSchedulers.mainThread()` indica lo scheduler che esegue le azioni sul Main Thread di Android, aggiornando l'interfaccia grafica visibile all'utente.

Nel precedente listato si è dato per scontato che l'observer passato al metodo `subscribe()` (riga 4) fosse già stato creato a parte; se ciò non fosse, di seguito viene presentata la porzione di codice completa (listato 3.13), la quale mostra sia la creazione di un oggetto `Observable` di stringhe, sia l'indicazione degli scheduler preposti allo svolgimento delle varie operazioni, sia la sottoscrizione con contestuale creazione dell'observer:

```
1  Observable.just("Alfa", "Beta", "Gamma", "Delta")
2          .subscribeOn(Schedulers.newThread())
3          .observeOn(AndroidSchedulers.mainThread())
4          .subscribe({ item -> Log.i("TAG", "onNext: $item") },     // onNext()
5                     { error -> Log.e("TAG", "onError: $error") }, // onError()
6                     { Log.i("TAG", "onComplete: All Done!") })    // onComplete()
```

Listato 3.13: Esempio completo di utilizzo di RxAndroid

Utilizzando insieme i metodi agili per la creazione di entità osservabili, gli scheduler messi a disposizione da RxKotlin 2 ed RxAndroid e le lambda expression, è stato possibile diminuire drasticamente la normale la quantità di codice necessaria.

3.4.3 Backpressure

La backpressure è una tecnica utilizzata con le entità osservabili di tipo `Flowable` e risolve il problema che si viene a creare nel momento in cui una sorgente di dati/eventi (observable) emette i suoi elementi ad una velocità superiore rispetto a quella che il suo sottoscrittore (observer) riesce a gestire, cioè superiore rispetto alla massima velocità con la quale l'osservatore riesce a processare i dati che gli arrivano. In questo caso serve una strategia, per esempio un buffer di ricezione abbastanza capiente dove raccogliere i dati in eccesso in attesa di essere processati, oppure ancora un metodo per decidere quando e quali elementi scartare in caso si rivelassero in sovrabbondanza.

Esistono diverse strategie di backpressure disponibili:

- `BackpressureStrategy.DROP`

  L'observer scarta gli elementi in eccesso che non riesce a processare.

- `BackpressureStrategy.BUFFER`

  L'observer predispone un buffer per raccogliere gli elementi in eccesso; li processerà man mano che finirà con i precedenti.

- `BackpressureStrategy.LATEST`

  L'observer predispone un buffer per raccogliere gli elementi in eccesso ma di questi processa solo il più recente, gli altri eventuali vengono scartati.

- `BackpressureStrategy.ERROR`

Viene lanciata una `MissingBackpressureException` in caso di presenza di elementi in eccesso.

- `BackpressureStrategy.MISSING`

  Non viene adottata alcuna particolare strategia di backpressure.

Di seguito un semplice esempio che mostra come possa essere utilizzata una strategia backpressure di tipo *drop* per stampare un milione di stringhe all'interno di una `TextView` a partire da un range di due milioni di interi (listato 3.14):

```
1  val textView = findViewById<TextView>(R.id.textView)
2
3  Flowable.range(1, 2_000_000)
4          .take(1_000_000)
5          .map { it.toString() }
6          .subscribeOn(Schedulers.computation())
7          .onBackpressureDrop()
8          .observeOn(AndroidSchedulers.mainThread())
9          .subscribe({ textView.text = it },
10                    { it.printStackTrace() },
11                    { textView.text = "Completed!" })
```

Listato 3.14: Esempio di utilizzo di una strategia di backpressure in RxKotlin 2

Dal precedente frammento di codice è evidente come RxKotlin 2 prediliga un approccio orientato alla programmazione funzionale, il che lo rende ancora più snello e leggibile; inoltre, è importante da tenere a mente come il corretto ordine di esecuzione dei metodi ed operatori rivesta un ruolo di primaria importanza per il risultato finale.

A riga 1 viene individuata e salvata la textview sulla quale verranno stampate le stringhe emesse dall'*upstream* (entità osservabile), alla riga 3 viene creato un `Flowable` che emette interi in ordine crescente da 1 a due milioni; tuttavia, nella riga successiva l'operatore `take` funge da filtro selezionando "solo" il primo milione di elementi emessi. A riga 5 avviene una trasformazione (operatore `map`) che viene eseguita su ogni elemento: ognuno di essi viene trasformato da intero a stringa. Tutte le precedenti operazioni sono a carico dello scheduler `Schedulers.computation()` (riga 6) e avvengono quindi in background. A riga 7 viene scelta la strategia di backpressure *drop*, ciò significa che se il *downstrem* (entità osservatrice) riuscirà a processare gli elementi alla stessa velocità alla quale li emette l'upstream allora il downstream riceverà e accetterà tutto il milione di stringhe, altrimenti dovrà scartare lungo il percorso alcuni elementi, quelli che non riuscirà a processare in tempo. La quantità di elementi scartati dall'observer sarà proporzionale al livello di congestione presente in ricezione. A riga 8 viene demandato l'aggiornamento continuo dell'interfaccia grafica al Main Thread dell'app Android in esecuzione, mentre a righe 9, 10 e 11 viene sottoscritto (e contestualmente creato) un observer che imposterà un nuovo testo nella `textView` quando riceverà una nuova stringa,

stampando lo *stack trace* dell'eccezione in caso di errore e la stringa '*Completed!*' in caso di completamento con successo.

Gli operatori `take` (riga 4) e `map` (riga 5) verranno descritti più nel dettaglio nella prossima sottosezione.

3.4.4 Operatori per i flussi di dati

Ogni implementazione reattiva (con qualsiasi linguaggio di programmazione) possiede specifici operatori per la manipolazione, il filtraggio, la combinazione ed altre operazioni sui flussi di dati osservabili. In questa sottosezione vengono elencati i principali presenti in RxKotlin 2, raggruppati per categoria. Inoltre, per ognuno di essi, viene anche mostrato il *marble diagram* che ne formalizza il comportamento.

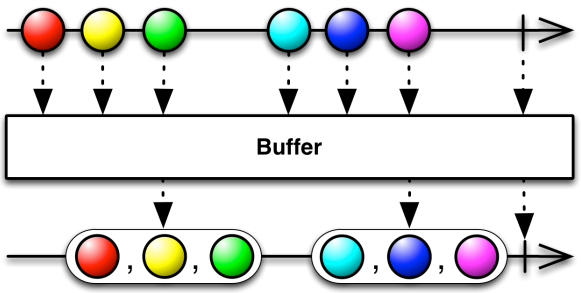
La maggior parte degli operatori lavora su un observable e restituisce un observable. Questo meccanismo permette di applicare gli operatori uno dopo l'altro, come in una catena. Ogni operatore nella catena modifica l'observable che riceve dall'operatore immediatamente precedente e lo invia poi a quello successivo, in cascata.

Operatori di trasformazione

Di seguito vengono elencati i principali operatori di trasformazione presenti in RxKotlin 2:

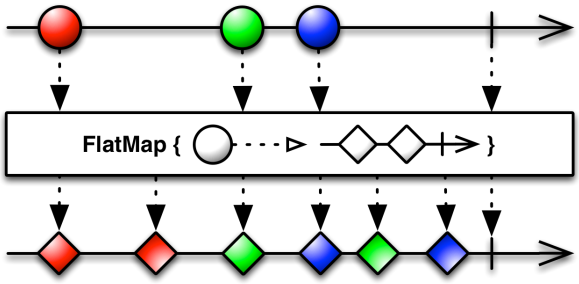
- `Buffer`

  Raccoglie periodicamente elementi da un observable e li emette in gruppi anziché emetterli uno alla volta:



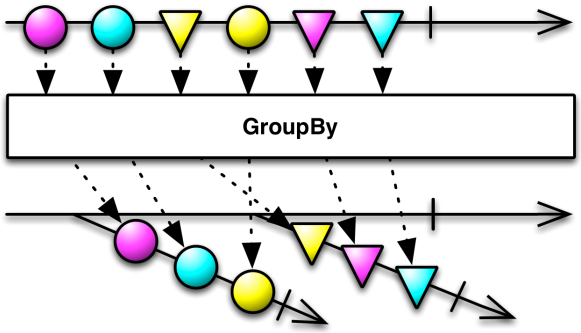
- `FlatMap`

  Trasforma gli elementi emessi da un observable in più flussi basandosi su una funzione passata come parametro; successivamente fonde i flussi in un unico observable:



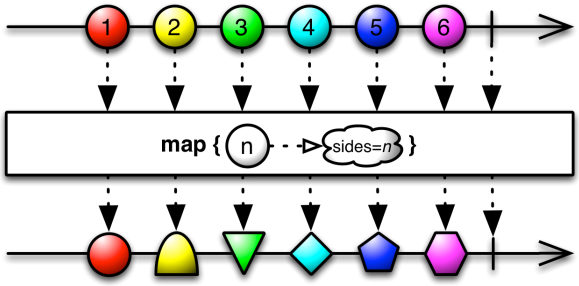
- `GroupBy`

  Divide un observable in un insieme di observable, i quali emettono ciascuno un sottoinsieme diverso di elementi appartenenti al flusso originario:



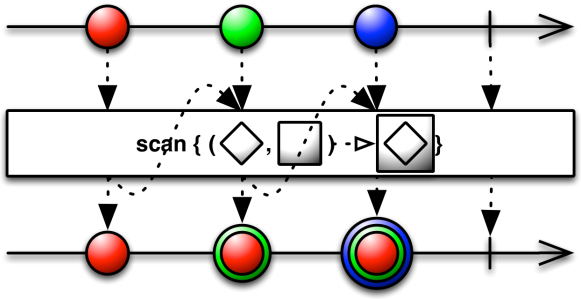
- `Map`

  Trasforma gli elementi emessi da un observable applicando una funzione a ciascuno di essi:



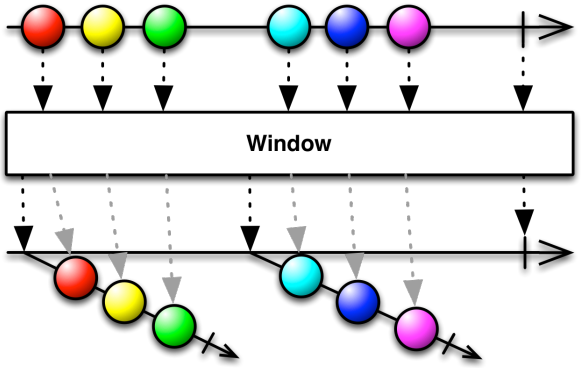
- `Scan`

  Applica una funzione a ciascun elemento emesso da un observable, in sequenza, ed emette ogni valore successivo:



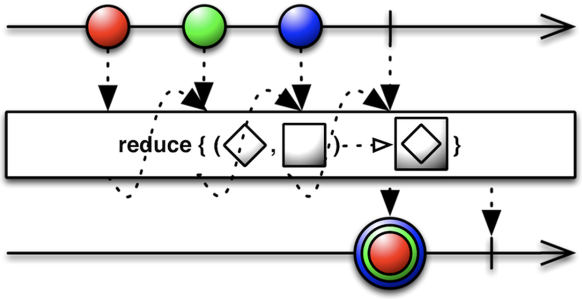
- `Window`

  Suddivide periodicamente gli elementi emessi da un observable in "finestre" osservabili ed emette ciascuna di esse all'interno di flussi separati, invece che emettere gli elementi uno alla volta lungo lo stesso flusso:



- `Reduce`

  Applica una funzione a ciascun elemento emesso da un observable, in sequenza, ed emette il valore finale:

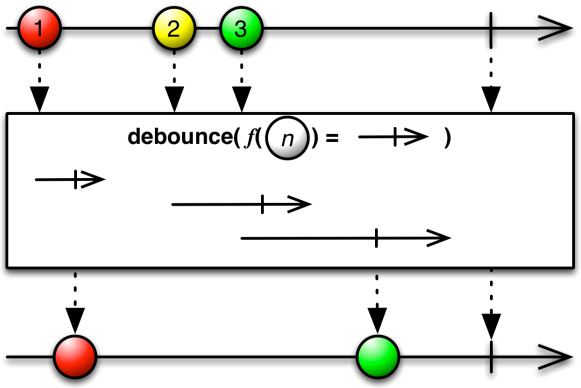


Operatori di filtraggio

Di seguito viene mostrato un elenco dei principali operatori di filtraggio presenti in RxKotlin 2:

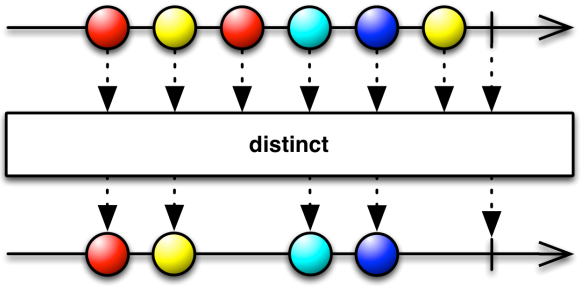
- `Debounce`

  Emette un elemento da un observable soltanto se in un determinato periodo di tempo non è stato emesso nessun altro valore:



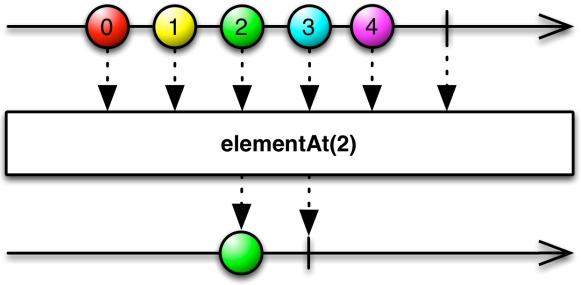
- `Distinct`

  Rimuove gli elementi duplicati emessi da un observable:



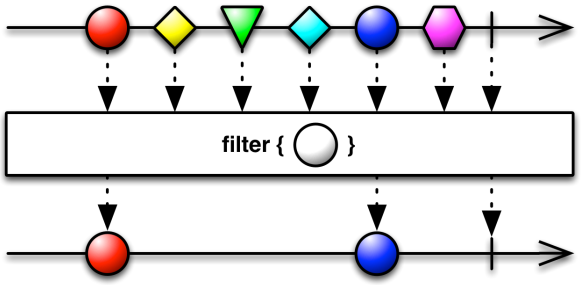
- `ElementAt`

  Emette solo l'elemento con indice specificato:



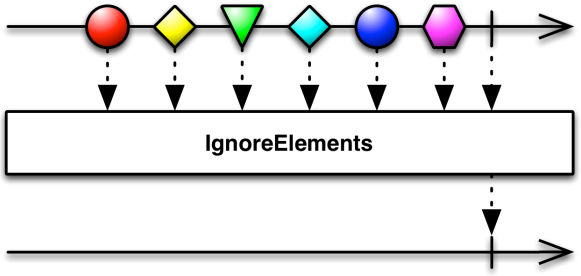
- `Filter`

  Emette solo quegli elementi che superano un test dei predicati specificato:



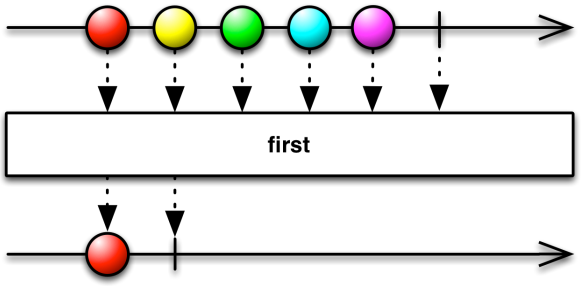
- `IgnoreElements`

  Ignora tutti gli elementi emessi dall'observable, richiamando soltanto `onComplete()` oppure `onError`:



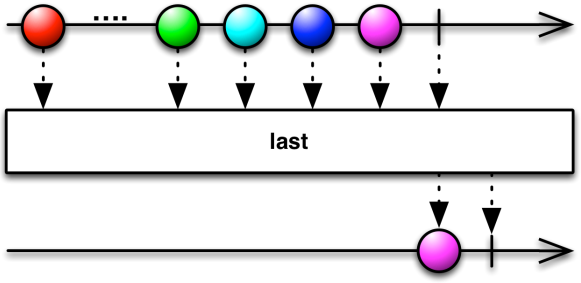
- `First`

  Emette solo il primo elemento emesso da un observable:



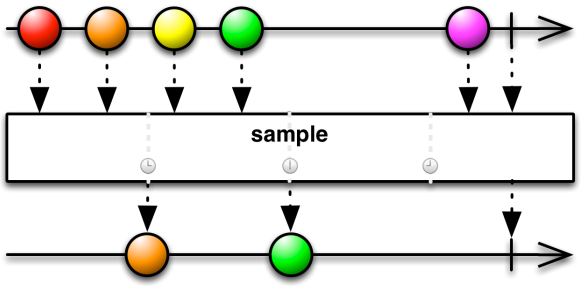
- `Last`

  Emette solo l'ultimo elemento emesso da un observable:



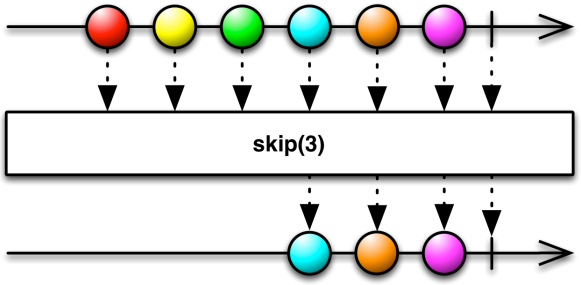
- `Sample`

  Emette l'elemento più recente emesso da un observable all'interno di intervalli di tempo prestabiliti:



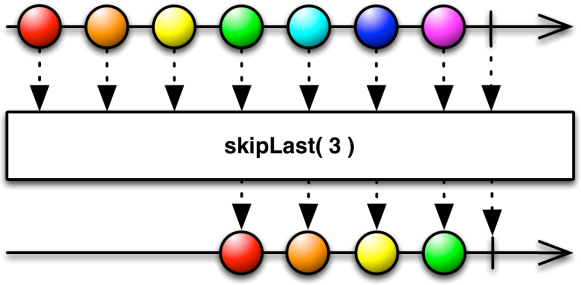
- `Skip`

  Rimuove i primi n elementi emessi da un observable:



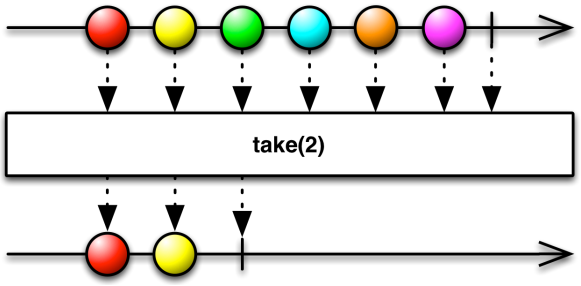
- `SkipLast`

  Rimuove gli ultimi n elementi emessi da un observable:



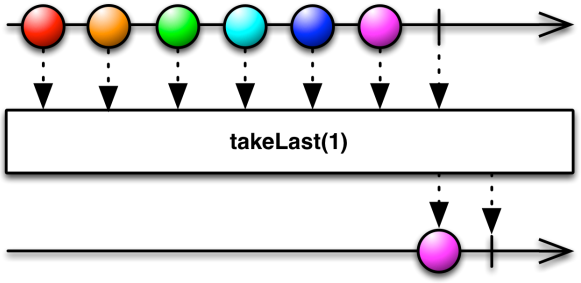
- `Take`

  Emette solo i primi n elementi emessi da un observable:



- `TakeLast`

  Emette solo gli ultimi n elementi emessi da un observable:

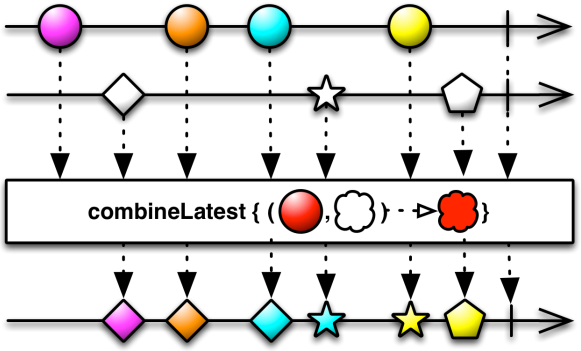


Operatori di combinazione

Di seguito vengono elencati i principali operatori di combinazione presenti in RxKotlin 2:

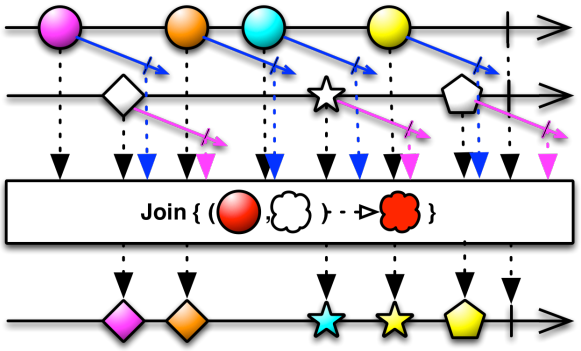
- `CombineLatest`

  Quando un elemento viene emesso da uno dei due observable, questo operatore combina l'ultimo oggetto emesso da ciascun flusso tramite una funzione specificata ed emette elementi in base ai risultati di quest'ultima:



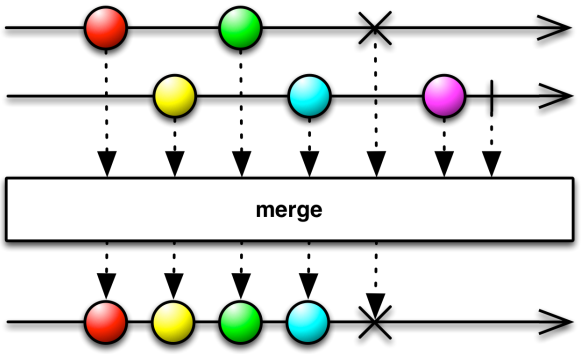
- `Join`

  Combina gli elementi emessi da due observable ogni volta che uno di essi viene emesso dal primo observable durante una finestra temporale definita in base a un elemento emesso dal secondo observable:



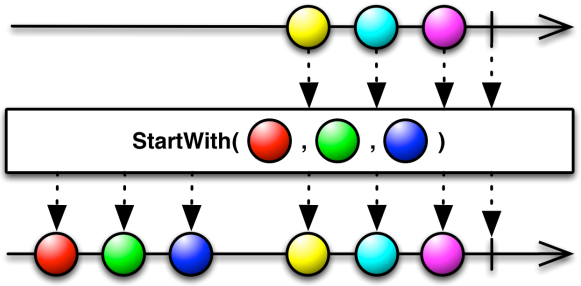
- `Merge`

  Unisce gli elementi emessi da due differenti observable in un unico flusso:



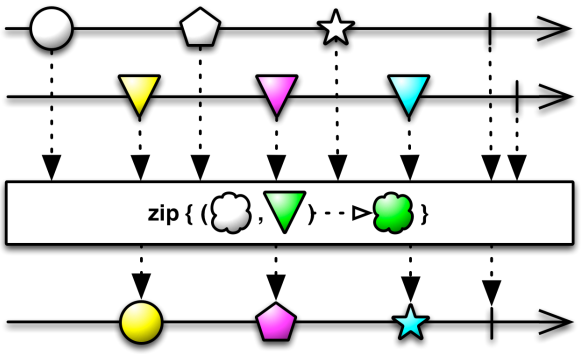
- `StartWith`

  Emette una sequenza specifica di elementi prima di iniziare ad emettere gli elementi emessi dall'observable:



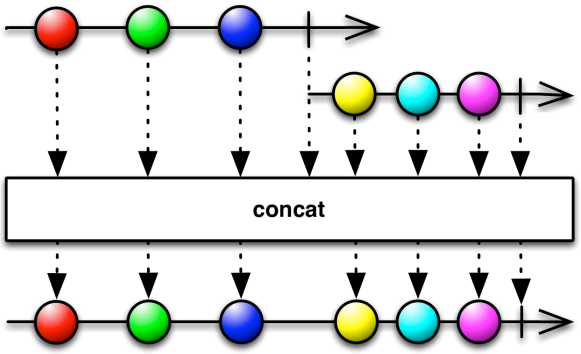
- `Zip`

  Combina le emissioni di più observable con una funzione specifica ed emette singoli elementi per ciascuna combinazione in base ai risultati di questa funzione:



- `Concat`

  Emette gli elementi di due o più observable senza attuare compenetrazione fra loro:

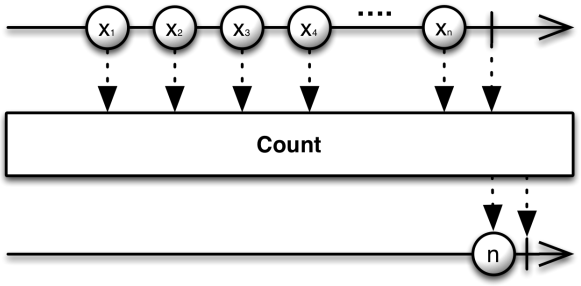


Altri operatori di utilità

Di seguito vengono elencati altri utili operatori presenti in RxKotlin 2:

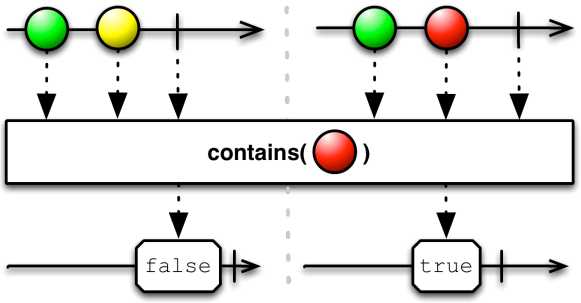
- `Count`

  Conta il numero di elementi emessi da un observable ed emette il valore calcolato:



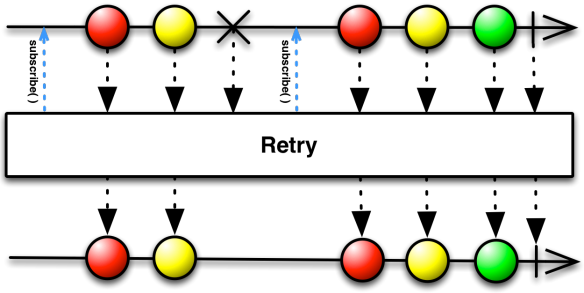
- `Contains`

  Determina se un observable emette un particolare elemento o meno:



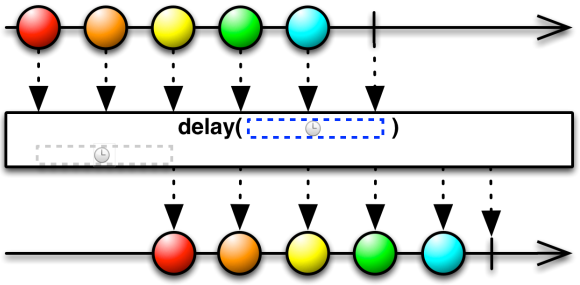
- `Retry`

  Se un observable emette un messaggio di errore, questo operatore tenta di sottoscriversi nuovamente ad esso nella speranza che questa volta non si verifichino errori:



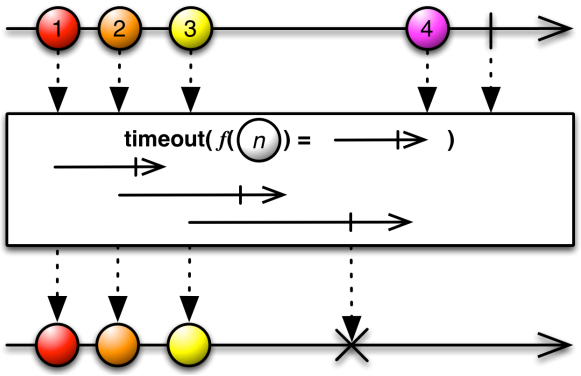
- `Delay`

  Sposta le emissioni di un observable in avanti nel tempo di un determinato incremento:



- `Timeout`

  Emette una notifica di errore se trascorre un determinato periodo di tempo senza elementi emessi da parte di un observable:



3.5 Confronto con le coroutines

Va innanzitutto premesso come non sia del tutto semplice confrontare RxKotlin e le coroutine: la prima è difatti una libreria di programmazione reattiva, mentre il secondo è un meccanismo offerto dal linguaggio Kotlin; tuttavia possono essere rilevate alcune somiglianze e differenze.

Entrambe le tecniche rese possibili da RxKotlin e dal meccanismo delle coroutine si propongono di affrontare l'asincronia: il primo sul piano della generazione ed elaborazione di flussi di dati asincroni, il secondo sulla scrittura di codice asincrono eseguibile in background. Le coroutine sono probabilmente più semplici da comprendere nel breve periodo, poiché non poggiano su concetti teorici comuni ad un paradigma di programmazione; sono tuttavia sicuramente più limitate a livello di funzionalità rispetto ad una libreria costruita su un'architettura basata ad eventi quale RxKotlin.

Risulta utile pensare a due situazioni; la prima nella quale un programmatore stia scrivendo codice sequenziale e abbia improvvisamente bisogno di richiamare un metodo asincrono: è plausibile che in questa situazione le coroutine siano più veloci e semplici da utilizzare, piuttosto che creare e "avvolgere" ogni elemento in observable. La seconda situazione, invece, vede il programmatore alle prese con la cattura e il controllo di varie sorgenti di dati che inviano informazioni in maniera asincrona (un esempio può essere un'applicazione Android): in questo caso può essere dispendioso e caotico l'utilizzo di coroutine, mentre può risultare vantaggioso utilizzare RxKotlin, in modo da gestire le sorgenti come flussi asincroni di dati osservabili e capaci di intercettare anche eventuali situazioni di errore.

Una somiglianza tra le coroutine ed RxKotlin per quanto riguarda la programmazione su sistemi Android, è che entrambi possono essere utilizzati per sostituire efficacemente un AsyncTask, necessitando pressappoco dello stesso impegno durante la stesura del codice.

Per quanto riguarda invece il grado di leggibilità, RxKotlin si adatta probabilmente meglio, anche in caso di compiti particolarmente complicati, situazione nella quale le coroutine tendono a rendere il codice non banale da comprendere e da gestire. RxKotlin promuove infatti un approccio funzionale ai problemi, concatenando le operazioni e mantenendo gli operatori "in linea"; le coroutine,

invece, non possiedono le stesse capacità e necessitano di ricorrere all'annidamento di più funzioni e all'indentazione. Inoltre, RxKotlin è sicuramente la scelta migliore quando si tratta di compiere trasformazioni complesse sui flussi di dati, possedendo un ricco insieme di operatori preposti a tale compito.

In conclusione, le coroutine ed RxKotlin dovrebbero essere utilizzati in base al contesto e ai quesiti da risolvere: in caso di piccoli problemi è consigliato utilizzare le coroutine, mentre nel caso di sistemi più complessi, con presenza di asincronia e di sorgenti di dati, RxKotlin rappresenta uno strumento più valido.

3.6 Test del codice

In RxKotlin 2 è possibile testare il codice tramite l'utilizzo della classe `TestObserver`. Viene di seguito mostrato subito un esempio (listato 3.15) e successivamente descritto nel dettaglio:

```
1  Observable.just(1, 2, 3)
2            .test()
3            .assertSubscribed()
4            .assertValueAt(1) { it == 2 }
5            .assertNever(4)
6            .assertValues(1, 2, 3)
7            .assertComplete()
8            .assertNoErrors()
```

Listato 3.15: Esempio di utilizzo degli operatori di test in RxKotlin 2

A riga 1 viene creato un observable che emette tre numeri interi. A riga 2 la funzione `test()` ritorna un oggetto appartenente alla classe `TestObserver`, il quale viene automaticamente sottoscritto all'observable appena creato. In questo modo è possibile richiamare alcuni operatori per compiere test sul flusso di dati: a riga 3 viene controllato che sia effettivamente avvenuta la sottoscrizione, mentre nelle tre righe successive si controllano gli elementi emessi (si controlla che l'elemento corrispondente all'indice 1 sia il numero intero 2, che non venga mai generato un 4 e che vengano generati tutti i numeri 1, 2 e 3). A riga 7 viene controllato l'avvenuto completamento delle emissioni di dati da parte del flusso osservabile, mentre a riga 8 viene verificato che non si siano riscontrati errori.

Prospettive future per RxKotlin su Android

RxKotlin, come specificato più volte all'interno della tesi, si basa principalmente sulla libreria RxJava, dalla quale eredita tutte le funzionalità e le caratteristiche distintive di un framework di programmazione reattiva, e su Kotlin, dal quale eredita invece il linguaggio. É dunque auspicabile che gli sviluppi futuri di RxKotlin dipenderanno da quelli che si verificheranno su RxJava e su Kotlin.

Kotlin è in continua evoluzione e viene soprattutto apprezzato dagli sviluppatori Android (dove è maggiormente impiegato), i quali si prevede che lo preferiranno di gran lunga a Java e lo utilizzeranno per la realizzazione della maggior parte delle nuove applicazioni[9] nel 2019 (figura 3.1):

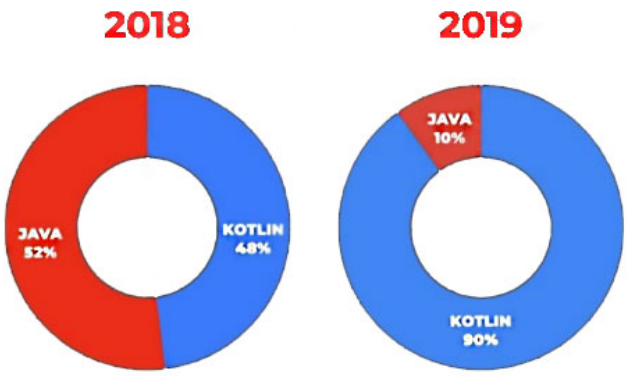


Figura 3.1: Previsione del numero di nuove app Android sviluppate con Java e con Kotlin nel 2018 e nel 2019. Immagine estratta da `https://www.jetbrains.com/`

Con l'uscita della versione 1.3 di Kotlin è inoltre molto probabile che le coroutine non saranno più in fase sperimentale[10], offrendo agli sviluppatori ulteriori strumenti affidabili per la programmazione asincrona.

Anche RxJava (e con esso RxKotlin) continuerà ad evolvere, introducendo probabilmente nuove funzionalità e operatori per la manipolazione dei flussi osservabili di dati e raggiungendo la terza versione della libreria.

[9]Per maggiori informazioni visitare: `https://www.jetbrains.com/`
[10]Per maggiori informazioni visitare: `http://archive.is/nHnEq`

Bibliografia

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx e Wolfgang de Meuter. «A survey on reactive programming». In: *ACM Computing Surveys* 45.4 (ago. 2013), pp. 1–34. DOI: 10.1145/2501654.2501666. URL: https://doi.org/10.1145/2501654.2501666.

[2] Gérard Berry e Georges Gonthier. «The Esterel synchronous programming language: design, semantics, implementation». In: *Science of Computer Programming* 19.2 (nov. 1992), pp. 87–152. DOI: 10.1016/0167-6423(92)90005-v. URL: https://doi.org/10.1016/0167-6423(92)90005-v.

[3] Frédéric Boussinot. «FairThreads: mixing cooperative and preemptive threads in C». In: *Concurrency and Computation: Practice and Experience* 18.5 (2006), pp. 445–469. DOI: 10.1002/cpe.919. URL: https://doi.org/10.1002/cpe.919.

[4] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem e Wolfgang De Meuter. «Loosely-Coupled Distributed Reactive Programming in Mobile Ad Hoc Networks». In: *Objects, Models, Components, Patterns*. Springer Berlin Heidelberg, 2010, pp. 41–60. DOI: 10.1007/978-3-642-13953-6_3. URL: https://doi.org/10.1007/978-3-642-13953-6_3.

[5] Gregory H. Cooper e Shriram Krishnamurthi. «Embedding Dynamic Dataflow in a Call-by-Value Language». In: *Programming Languages and Systems*. Springer Berlin Heidelberg, 2006, pp. 294–308. DOI: 10.1007/11693024_20. URL: https://doi.org/10.1007/11693024_20.

[6] Antony Courtney. «Frappé: Functional Reactive Programming in Java». In: *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, 2001, pp. 29–44. DOI: 10.1007/3-540-45241-9_3. URL: https://doi.org/10.1007/3-540-45241-9_3.

[7] Evan Czaplicki e Stephen Chong. «Asynchronous functional reactive programming for GUIs». In: *ACM SIGPLAN Notices* 48.6 (giu. 2013), p. 411. DOI: 10.1145/2499370.2462161. URL: https://doi.org/10.1145/2499370.2462161.

[8] Jonathan Edwards. «Coherent reaction». In: *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*. ACM Press, 2009. DOI: 10.1145/1639950.1640058. URL: https://doi.org/10.1145/1639950.1640058.

[9] Conal M. Elliott. «Push-pull functional reactive programming». In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell - Haskell '09*. ACM Press, 2009. DOI: 10.1145/1596638.1596643. URL: https://doi.org/10.1145/1596638.1596643.

[10] Conal Elliott e Paul Hudak. «Functional reactive animation». In: *ACM SIGPLAN Notices* 32.8 (ago. 1997), pp. 263–273. DOI: 10.1145/258949.258973. URL: https://doi.org/10.1145/258949.258973.

[11] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. «Design Patterns: Abstraction and Reuse of Object-Oriented Design». In: *ECOOP' 93 — Object-Oriented Programming*. Springer Berlin Heidelberg, pp. 406–431. DOI: 10.1007/3-540-47910-4_21. URL: https://doi.org/10.1007/3-540-47910-4_21.

[12] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring e M. Trakhtenbrot. «STATEMATE: a working environment for the development of complex reactive systems». In: *IEEE Transactions on Software Engineering* 16.4 (apr. 1990), pp. 403–414. DOI: 10.1109/32.54292. URL: https://doi.org/10.1109/32.54292.

[13] Paul Hudak, Antony Courtney, Henrik Nilsson e John Peterson. «Arrows, Robots, and Functional Reactive Programming». In: *Advanced Functional Programming*. Springer Berlin Heidelberg, 2003, pp. 159–187. DOI: 10.1007/978-3-540-44833-4_6. URL: https://doi.org/10.1007/978-3-540-44833-4_6.

[14] Cor J. Kalkman. «LabVIEW: A software system for data acquisition, data analysis, and instrument control». In: *Journal of Clinical Monitoring* 11.1 (gen. 1995), pp. 51–58. DOI: 10.1007/bf01627421. URL: https://doi.org/10.1007/bf01627421.

[15] Joeri De Koster, Tom Van Cutsem e Wolfgang De Meuter. «43 years of actors: a taxonomy of actor models and their key properties». In: *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2016*. ACM Press, 2016. DOI: 10.1145/3001886.3001890. URL: https://doi.org/10.1145/3001886.3001890.

[16] Chris Lattner e Vikram Adve. «The LLVM Compiler Framework and Infrastructure Tutorial». In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 15–16. DOI: 10.1007/11532378_2. URL: https://doi.org/10.1007/11532378_2.

[17] Henry Lee e Eugene Chuvyrov. «Reactive Extensions for.NET». In: *Beginning Windows Phone 7 Development*. Apress, 2010, pp. 383–411. DOI: 10.1007/978-1-4302-3217-9_18. URL: https://doi.org/10.1007/978-1-4302-3217-9_18.

[18] Andrea Maglie. «ReactiveX and RxJava». In: *Reactive Java Programming*. Apress, 2016, pp. 1–9. DOI: 10.1007/978-1-4842-1428-2_1. URL: https://doi.org/10.1007/978-1-4842-1428-2_1.

[19] Ingo Maier e Martin Odersky. «Higher-Order Reactive Programming with Incremental Lists». In: *ECOOP 2013 – Object-Oriented Programming*. Springer Berlin Heidelberg, 2013, pp. 707–731. DOI: 10.1007/978-3-642-39038-8_29. URL: https://doi.org/10.1007/978-3-642-39038-8_29.

[20] Sean McDirmid e Wilson C. Hsieh. «SuperGlue: Component Programming with Object-Oriented Signals». In: *ECOOP 2006 – Object-Oriented Programming*. Springer Berlin Heidelberg, 2006, pp. 206–229. DOI: 10.1007/11785477_15. URL: https://doi.org/10.1007/11785477_15.

[21] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield e Shriram Krishnamurthi. «Flapjax». In: *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA 09*. ACM Press, 2009. DOI: 10.1145/1640089.1640091. URL: https://doi.org/10.1145/1640089.1640091.

[22] Marcin Moskala e Igor Wojda. *Android Development with Kotlin*. Birmingham: Packt Publishing, ago. 2017. ISBN: 9781787123687.

[23] Henrik Nilsson, Antony Courtney e John Peterson. «Functional reactive programming, continued». In: *Proceedings of the ACM SIGPLAN workshop on Haskell - Haskell '02*. ACM Press, 2002. DOI: 10.1145/581690.581695. URL: https://doi.org/10.1145/581690.581695.

[24] Balaji Varanasi. *Introducing Gradle*. Apress, 2015. DOI: 10.1007/978-1-4842-1031-4. URL: https://doi.org/10.1007/978-1-4842-1031-4.

[25] Zhanyong Wan e Paul Hudak. «Functional reactive programming from first principles». In: *ACM SIGPLAN Notices* 35.5 (mag. 2000), pp. 242–252. DOI: 10.1145/358438.349331. URL: https://doi.org/10.1145/358438.349331.