View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by AMS Tesi di Laurea

Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE - CAMPUS DI CESENA

Corso di Laurea in Ingegneria e Scienze Informatiche

Indagine sull'utilizzo di Scala per progetti Android

Elaborato in
Programmazione ad Oggetti

Relatore:
Prof. Mirko VIROLI

Presentata da:
Giuseppe Ettore RADAELLI

Correlatore:
Dott. Roberto CASADEI

Sessione III
Anno Accademico 2015/2016

Alla piccola Elisa,
ed a chi, nonostante tutto, continua
a su(o)pportarmi ...

Indice

Introduzione

Android, il sistema operativo sviluppato da Google e lanciato per la prima volta nel 2008, è oggigiorno eseguito su milioni di dispositivi diversi, quali telefoni cellulari, tablet, TV, etc. Diversamente dai suoi concorrenti principali, poggia le sue basi su una struttura open-source basata su Linux Kernel, rendendolo il sistema operativo ormai più diffuso al mondo. Grazie a questa sua natura, Android risulta molto flessibile per lo sviluppo di software ad esso dedicato.

Le applicazioni Android sono in genere sviluppate con il linguaggio di programmazione Java ed il Software Development Kit di Android (Android SDK). Compilando il codice Java si ottiene del JVM bytecode, che in una fase successiva viene convertito in un formato specifico Android, quest'ultimo sarà poi pronto per l'esecuzione sul Runtime Android (ART), ossia l'implementazione JVM di Android.

Questo processo, che accade interamente dietro le quinte, non ci riguarda, ciò ci dovrebbe lasciare la libertà di decidere come generare il nostro bytecode, nonostante Java sia l'unico linguaggio ufficialmente supportato e documentato per la creazione di app Android. Negli ultimi anni si stanno diffondendo sempre più i linguaggi che possono essere compilati in bytecode eseguibile su una JVM. Non tutti però possono essere utilizzati sulla piattaforma Android, chi per motivi di compatibilità, chi di prestazioni. Tra questi, ce n'è sono tre che hanno attirato di più l'attenzione degli sviluppatori: Groovy, Kotlin e Scala.

L'obiettivo principale di questa tesi consiste proprio nello studio di questa

possibilità, cercando di capire se sia possibile la realizzazione di app Android utilizando un linguaggio diverso da Java, in particolare, nel nostro caso, utilizzeremo Scala.
Ma perché proprio Scala? In primis perché è uno dei linguaggi la cui compilazione del codice sorgente produce Java bytecode per l'esecuzione su una JVM. Inoltre è un linguaggio ibrido, oltre alla programmazione ad oggetti, supporta anche quella funzionale: permette alle funzioni di essere annidate, implementa un sistema leggero per la dichiarazione di funzioni anonime e molto altro. In aggiunta copre la maggior parte delle caratteristiche di Java e aggiunge una serie di funzionalità molto utili, come l'inferenza dei tipi, tratti etc.
Una volta raggiunto il primo obiettivo, il passo successivo sarà cercare di capire se sia possibile integrare il framework Scafi in un progetto Android. Questa piattaforma, sviluppata in Scala, offre tutta una serie di primitive per programmare il comportamento di un sistema in modo aggregato, che approfondiremo meglio negli altri capitoli.
Riassumendo, questo documento è strutturato in quattro parti:

1. introduzione ad Android;
2. panoramica su Scala con particolare attenzioni alle differenze rispetto a Java ed alle novità che introduce;
3. come prepararsi l'ambiente di sviluppo Scala su Android;
4. integrazione di Scafi in Android.

Capitolo 1

Android

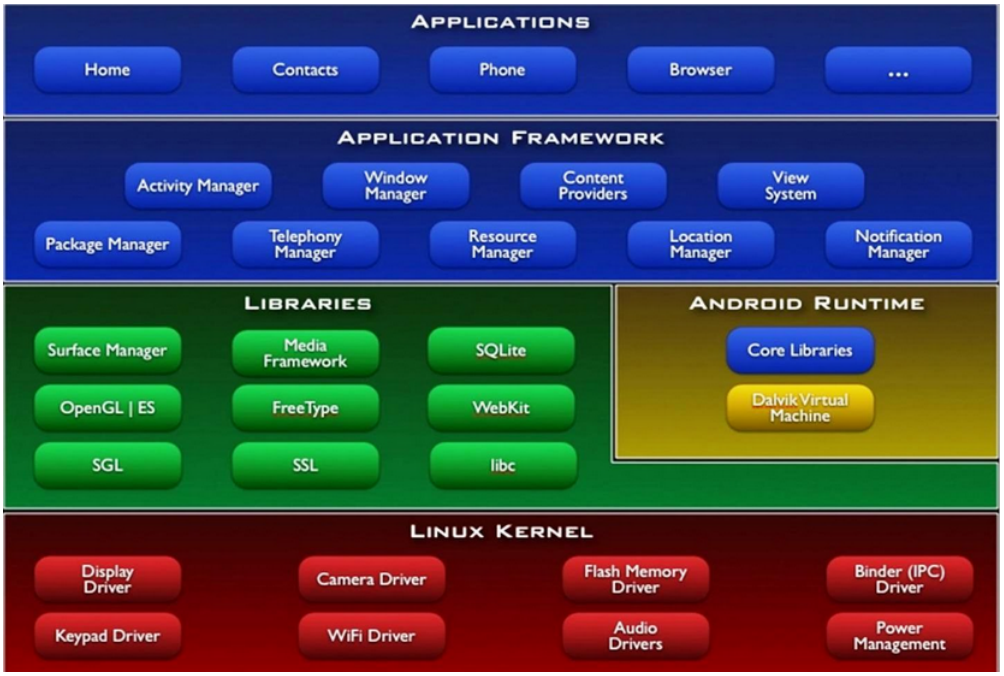
Android[1] è un sistema operativo per dispositivi mobili sviluppato da Google Inc. e basato su kernel Linux, non contiene codice GNU, pertanto non è da considerarsi una distribuzione GNU/Linux per sistemi embedded. È stato progettato principalmente per smartphone e tablet, con interfacce utente specializzate per televisori (Android TV), automobili (Android Auto), orologi da polso (Android Wear) e occhiali (Google Glass).
Android non esegue bytecode Java, quindi non ha bisogno di una JVM, per ottimizzare al massimo l'utilizzo delle risorse dei dispositivi, originariamente

[1] `https://www.android.com/intl/it_it/`

era stata creata una propria Virtual Machine che prende il nome di Dalvik (nome delle città di origine dello sviluppatore). Successivamente, dalla versione di Android 5.0 Lollipop, è stato sostituito da ART (Android Run-Time).

1.1 Architettura

L'architettura Android è organizzata a strati dove i livelli inferiori offrono servizi a quelli superiori dando un più alto grado di astrazione.



- **Linux Kernel**
  rappresenta il livello più basso, la necessità era infatti quella di disporre di un vero e proprio sistema operativo che fornisse gli strumenti di basso livello per l'astrazione dell'hardware sottostante con i driver per la gestione delle diverse periferiche.

- **Librerie native**
  appena sopra il kernel, sono realizzate in C e C++ e rappresentano il

cuore vero e proprio di Android. Si tratta di librerie che fanno riferimento a un insieme di progetti open source utilizzate da vari componenti del sistema. Tra queste troviamo:

- *Surface Manager*, che gestisce le View;
- *OpenGL ES*, utilizzata per la grafica 3D;
- *SGL*, costituisce il motore grafico di Android;
- *Media Framework*, serve per gestire la multimedialità;
- *FreeType*, serve per la gestione dei font;
- *SQLite*, che permette la memorizzare di informazioni in modo persistente sul dispositivo;
- *Webkit*, browser integrato nella piattaforma;
- *SSL*, libreria per la gestione dei Secure Socket Layer;
- *Libc*, implementazione della libreria standard C libc ottimizzata per dispositivi basati su Linux.

- **Android Runtime**
  costituito da una libreria core che permette l'esecuzione delle applicazioni e da una virtual machine (Dalvik o ART).

- **Application Framework**
  insieme di API e componenti per l'esecuzione di funzionalità ben precise e di fondamentale importanza in ciascuna applicazione Android. Tra queste troviamo:

  - *Activity Manager*, con il quale quale l'utente interagisce con l'applicazione;
  - *Window Manager*, che gestisce le finestre delle varie applicazioni;
  - *Content Providers*, che condivide le informazioni tra i vari processi;
  - *View System*, che gestisce gli elementi grafici (view) utilizzati nella costruzione dell'interfaccia utente (UI);

- *Package Manager*, che gestisce i processi di installazione e rimozione delle applicazioni dal sistema;
- *Telephony Manager*, che consente di accedere alle informazioni e all'utilizzo dei servizi di telefonia del dispositivo;
- *Resource Manager*, con compito preciso di ottimizzare le risorse;
- *Location Manager*, API per la gestione delle informazioni relative alla localizzazione;
- *Notification Manager*, che gestisce le informazioni di notifica tra il dispositivo e l'utente.

- **Application**
  applicazioni utilizzate dall'utente, come per esempio il telefono, il calendario o la rubrica.

1.2 Dalvik vs JVM

Le macchine virtuali (VM) offrono un alto livello di astrazione sopra il sistema operativo nativo, permettendo di avere la stessa piattaforma di esecuzione indipendentemente dal sistema su cui sono installate o dall'architetture hardware del dispositivo.
Che cosa dovrebbe implementare in genere una macchina virtuale? Essa dovrebbe emulare le operazioni effettuate da una CPU fisica e comprendere i seguenti concetti:

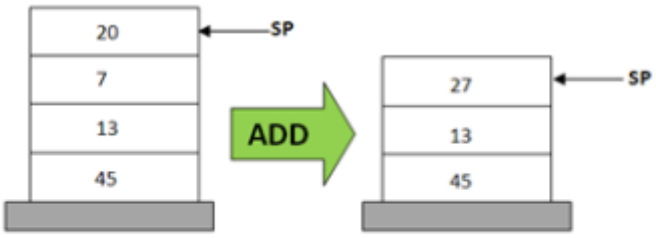
- compilazione della lingua sorgente in bytecode specifico per la VM;
- strutture dati che contengano le istruzioni e gli operandi;
- stack per le operazioni di chiamata di funzione;
- Instruction Pointer (IP) che punta alla prossima istruzione da eseguire;

- CPU virtuale, il cui Instraction Dispatcher deve recupera l'istruzione successiva (indirizzata dall'IP), decodificare gli operandi ed eseguire l'istruzione.

Ci sono fondamentalmente due modi per implementare una macchina virtuale:

- **Stack-based** (JVM);
- **Register-based** (Dalvik).

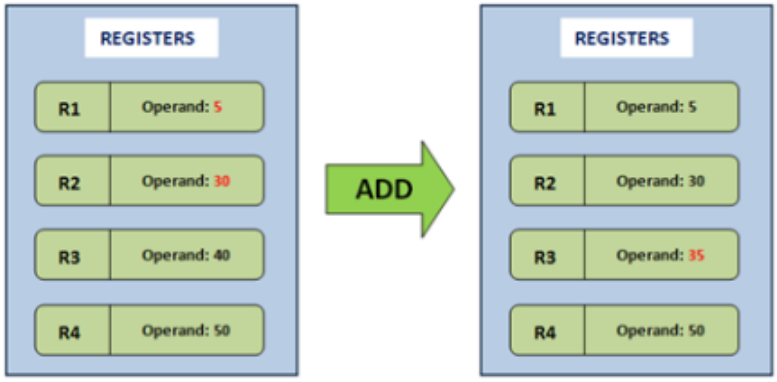
La differenza tra i due approcci sta nel meccanismo di archiviazione e recupero dei loro operandi e dei loro risultati.
La struttura di memoria, cioè dove vengono archiviati gli operandi, di una macchina virtuale Stack-based è organizzata come una pila (stack). In questo caso il pop (estrazione) ed il push (memorizzazione) dei dati avviene in modalità LIFO (Last In First Out). Un esempio dell'operazione di somma è mostrato nell'esempio sottostante (dove 20, 7 e "result" sono gli operandi).



1. **POP 20**
2. **POP 7**
3. **ADD 20, 7, result**
4. **PUSH result**

Nelle VM Stack-based, tutte le operazioni di aritmetica e logica sono effettuate tramite il push ed il pop degli operandi e dei risultati nello stack. Un vantaggio di questo modello è che gli operandi vengono indirizzati implicitamente dal puntatore dello stack (SP). Ciò significa che per la macchina virtuale non è necessario conoscere i loro indirizzi in modo esplicito.

Nell'implementazione di una macchina virtuale Register-based, la struttura dati dove vengono archiviati gli operandi è basata sui registri della CPU. Qui le istruzioni devono contenere gli indirizzi (i registri) degli operandi, vale a dire gli operandi sono esplicitamente indirizzati nell'istruzione. Ad esempio, per un'operazione di addizione, la sequenza di istruzioni sarebbe più o meno come segue.



1. **ADD R1, R2, R3 ;** # Add contents of R1 and R2, store result in R3

A differenza dell'altra architettura, gli indirizzi degli operandi (R1, R2 e R3) devono essere menzionati in modo esplicito. Il vantaggio che si ha, è che l'overhead causato dai vari push e pop nella pila è inesistente e quindi le istruzioni vengono eseguite più velocemente. Quest approccio permette anche alcune ottimizzazioni, per esempio il risultato di espressioni comuni nel codice può essere calcolato una volta, memorizzato in un registro e riusato in un secondo momento, questo riduce il costo di ricalcolare l'espressione. Uno svantaggio di questo modello consiste nel fatto che le sue istruzioni, dato che bisogna specificare gli indirizzi in modo esplicito, sono più lunghe di un'istruzione di un modello Stack-based.

1.3 Da Dalvik ad ART

Originariamente Android era stato creato una propria Virtual Machine che prende il nome di Dalvik (nome delle città di origine dello sviluppatore), ma

con la versione di Android 5.0 Lollipop, questa viene sostituita da ART (Android Run-Time). In questo paragrafo scopriremo le differenza tra le due, senza soffermarci nei dettagli.
La caratteristica principale della Dalvik è la compilazione JIT (Just-In-Time), questo tipo di compilazione funziona in questo modo:

1. si scarica un file `.apk` dal Playstore o da un qualsiasi altro store compatibile;

2. una volta installato si crea un file `.dex` contenente il bytecode (ossia un linguaggio intermedio più astratto tra il linguaggio macchina e il linguaggio di programmazione);

3. durante l'esecuzione di un programma la Dalvik si occupa di tradurre il bytecode in linguaggio macchina così da divenire eseguibile

4. una volta terminata l'operazione il codice tradotto viene perso e alla prossima esecuzione del programma si ricomincia dal punto 1.

Uno dei lati positivi della compilazione JIT è che non richiede una grande quantità in termini di memoria del dispositivo, c'è da dire, però, che la compilazione in fase d'esecuzione, sforzando il processore, ne limita l'autonomia in termini di batteria per il dispositivo.
Dalla versione 4.4 KitKat è stata introdotta una nuova runtime chiamata ART in via del tutto sperimentale, divenuta, poi, nella versione 5.0 Lollipop, la runtime predefinita. ART è progettata per essere completamente compatibile con il formato di bytecode esistente di Dalvik. Come la Dalvik anche ART ha bisogno di compilare il codice contenuto nel file `.apk` dopo la sua installazione ma affronta l'argomento utilizzando un approccio AOT (Ahead-of-Time):

1. si scarica un file `.apk` dal Playstore o da un qualsiasi altro store compatibile;

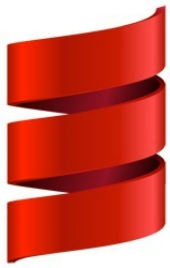
2. al momento dell' installazione l' applicazione viene anche compilata creando un file `.ELF` eseguibile

3. una volta avviata l'applicazione, il codice sarà già pronto per essere eseguito ed al termine dell' esecuzione rimarrà tale.

ART ha moltissimi vantaggi rispetto alla Dalvik in quanto, non dovendo compilare ogni volta il file `.dex`, l'esecuzione di un applicazione e il sistema operativo stesso risulterà più fluido e performante rispetto a prima, dato che il processore sarà occupato solo dall'esecuzione e non dalla compilazione, a dispetto però di una maggior lentezza durante l'installazione, visto che è comprensiva di compilazione e nella memoria occupata dall'applicazione che risulterà maggiore rispetto all'utilizzo della Dalvik. Oltre a maggiori prestazioni l'autonomia stessa del dispositivo ne gioverà visto il minor utilizzo della CPU durante l'esecuzione. Inoltre è dotata di una modalità MultiDex, su cui ci soffermeremo successivamente.

Capitolo 2

Scala



Scala[1]: acronimo di "SCAlable LAnguage", è un linguaggio di programmazione multi-paradigma moderno, progettato per esprimere pattern comuni di programmazione in modo conciso, elegante e type-safe. Inoltre integra agevolmente le caratteristiche dei linguaggi orientati agli oggetti e funzionali. Lo origioni di Scala, risalgono al 2001 ad opera di Martin Odersky, già autore dell'attuale compilatore javac e uno dei padri dei generics di Java, insieme ai ricercatori dell'Ecole Polytechnique Fèdèrale de Lausanne (EPFL). La prima versione non pubblica si ebbe nel 2003, solo un anno dopo si annunciò al mondo, prima su piattaforma Java e successivamente su quella .NET. Quello che colpisce di Scala, per lo sviluppatore Java, è la possibilità di avere un nuovo linguaggio capace di sfruttare le prestazioni incredibili della JVM

[1] `https://www.scala-lang.org/`

e l'abbondanza delle librerie Java che sono state sviluppate per oltre dieci anni.

2.1 Tipizzazione

Tipizzazione statica o tipizzazione dinamica? E' questa la prima delle scelte fondamentali che deve compiere chi sta per progettare un nuovo linguaggio. Per esempio Scala e Java sono linguaggi staticamente tipati, mentre Ruby, Python, Groovy, JavaScript e Smalltalk sono linguaggi dinamicamente tipati.
In poche parole, possiamo dire che nella tipizzazione statica una variabile è legata a un particolare tipo per tutto il proprio tempo di vita. Il tipo della variabile non può essere cambiato e la variabile può fare riferimento solo a istanze di tipo compatibile con il proprio; cioè, se una variabile fa riferimento a un valore di tipo A, non potete assegnarle un valore di un tipo B differente a meno che B non sia un sottotipo di A.
Nella tipizzazione dinamica il tipo è legato al valore, non alla variabile. Quindi, una variabile può fare riferimento a un valore di tipo A e successivamente vedersi assegnare un valore di tipo X non correlato ad A.
Si usa il termine dinamicamente tipato perché il tipo di una variabile viene valutato quando essa viene usata durante l'esecuzione, mentre in un linguaggio staticamente tipato il tipo viene valutato durante la compilazione. Questa potrebbe sembrare una piccola differenza, ma ha un impatto profondo sulla filosofia, sulla progettazione e sulla implementazione di un linguaggio.
Essendo staticamente tipato Scala ha un meccanismo di Type Inference integrato, che permette al programmatore di omettere alcune annotazioni di tipo. All'interno di applicazioni spesso non è necessario specificare il tipo di una variabile, poiché il compilatore può dedurre il tipo dall'espressione di inizializzazione della variabile. Anche i tipi di ritorno dei metodi possono spesso essere omessi in quanto corrispondono al tipo di corpo, che viene dedotto sempre dal compilatore.

2.2 Paradigma misto: OOP + FP

Come accennato precedentemente Scala supporta appieno i due paradigmi di programmazione, orientata agli oggetti (OOP) e funzionale (FP).
Scala migliora il supporto OOP di Java con l'aggiunta dei tratti, un modo pulito di implementare le classi usando la composizione dei mixin, si può pensare ai tratti come all'unificazione delle interfacce con la loro implementazione (ma su questo ci soffermeremo successivamente). Scala è un linguaggio orientato agli oggetti "puro" nel senso che ogni cosa è un oggetto, inclusi i numeri e le funzioni. In questo differisce da Java che invece distingue tra tipi primitivi (come `boolean` e `int`) e tipi referenziati, inoltre, Java non permette la manipolazione di funzioni come fossero valori. Poiché i numeri sono oggetti, hanno dei metodi. Di fatti un'espressione aritmetica come la seguente:

```
1 + 2 * 3 / x
```

consiste esclusivamente di chiamate a metodi e risulta equivalente alla seguente espressione:

```
(1).+(((2).*(3))./(x))
```

Questo significa anche che +, *, etc... sono identificatori validi in in Scala, che consente di usare nomi di metodo non alfanumerici: potete chiamare i metodi +, -, $, o in qualunque modo desideriate. Similmente, un metodo senza argomenti può essere invocato senza il punto. Questa notazione viene chiamata "postfissa".
Tuttavia, per ottimizzare le prestazioni, Scala usa i tipi primitivi della macchina virtuale sottostante ogni volta che è possibile. In più, non sono supportati membri "statici" o a livello di classe per i tipi, dato che non sono effettivamente associati ad un'istanza. Invece, Scala supporta un costrutto di oggetto singleton per i casi in cui è necessaria una sola istanza di un tipo.
La FP è un paradigma di programmazione più vecchio della OOP, ma l'in-

teresse per essa sta aumentando a causa del modo in cui semplifica certi problemi di progettazione, in particolare quelli legati alla concorrenza. I linguaggi funzionali "puri" non permettono alcuno stato mutabile, evitando di conseguenza il bisogno di sincronizzazione sull'accesso condiviso allo stato mutabile. Invece, i programmi scritti in linguaggi puramente funzionali comunicano scambiando messaggi tra processi autonomi e concorrenti. Scala supporta questo modello con la sua libreria di attori (anche di questi parleremo successivamente), ma consente di usare variabili mutabili e immutabili. Le funzioni possono essere assegnate a variabili o passate ad altre funzioni, esattamente come gli altri valori. Questa caratteristica promuove la composizione di comportamenti avanzati usando operazioni primitive. Dato che Scala aderisce al principio per cui ogni cosa è un oggetto, in Scala anche le funzioni sono oggetti.

2.3 Sintassi

Nel seguente paragrafo saranno descritti gli aspetti sintattici più importanti di Scala come la definizione di variabili, classi e tratti. Da notare che in Scala l'uso del punto e virgola come terminatore di linea di codice è a discrezione del programmatore perché il compilatore è in grado di determinare automaticamente la fine della riga come la fine dell'istruzione. Scala si comporta come molti linguaggi di scripting che trattano la fine di una riga come la fine di un'struzione o di un'espressione. Quando le istruzioni o le espressioni sono troppo lunghe per stare su una sola riga, di solito Scala è in grado di dedurre se state proseguendo sulla riga successiva.

2.3.1 Variabili e costanti

La definizione di variabili e costanti in Scala avviene mediante l'uso di due parole chiave:

- `val`
  usato per dichiarare una "variabile" immutabile, ossia di sola lettu-

ra, deve essere inizializzata, cioè definita, nel momento in cui viene dichiarata;

- `var`
  usato per dichiarare una variabile mutabile, di lettura e scrittura.

In generale per poter dichiarare una variabile si usa la seguente notazione:

```
var/var Nome_Variabile : Tipo = [Valore Iniziale]
```

Ecco un paio di esempi:

```
var nome : String = "Giuseppe"
val eta : Int = 29
```

Quando si assegna un valore iniziale ad una variabile il compilatore di Scala riesce ad individuare il tipo di variabile dichiarata in base al valore assegnato, questo perché Scala supporta la type inference (inferenza di tipo). In altre parole, il compilatore del linguaggio può dedurre una certa quantità di informazioni di tipo dal contesto, senza che vi siano annotazioni di tipo esplicite. Per questo motivo gli esempi precedenti possono essere riscritti come segue:

```
var nome = "Giuseppe"
val eta = 29
```

2.3.2 Classi e metodi

Come visto precedentemente Scala è un linguaggio orientato agli oggetti e come tale presenta il concetto di classe.
Di seguito si presenta un semplice esempio per capire in che modo avviene la definizione delle classi in Scala. La seguente classe definisce due attributi d'istanza `x` e `y` e un metodo `move`, il quale non ritorna alcun valore. Il nome della classe svolge la funzione di costruttore a cui vengono passati un certo numero di parametri: nel nostro esempio tali parametri sono x1 e y1:

```
class Point(x1: Int, y1: Int) {

    var x: Int = x1
    var y: Int = y1

    def move(dx: Int, dy: Int) {
        x = x + dx
        y = y + dy
    }
}
```

La definizione delle funzioni avviene secondo la seguente sintassi:

```
def nomeFunzione ([lista di parametri]) : [return tipo] = {
    corpo della funzione
    return [espressione]
}
```

In Scala non possiamo avere classi statiche, per'øsi hanno oggetti singleton. Un Singleton è una classe che può avere una sola istanza, si crea utilizzando la parola chiave `object` al posto della parola chiave `class`. Dal momento che non è possibile creare un'istanza di un oggetto Singleton, non è possibile passare parametri al costruttore principale. Di seguito è riportato un esempio:

```
object Test {

    def main(args: Array[String]) {

        val point = new Point(10, 20)
        printPoint

        def printPoint{
```

```
            println ("Point x location : " + point.x)
            println ("Point y location : " + point.y)
        }
    }
}
```

2.3.3 Tratti

L'editarietà multipla da la possibilità ad una classe di possedere più superclassi, permettendo quindi di ottenere classi con più comportamenti e favorendo un maggior riutilizzo del codice. Nonostante ciò, si rituene che l'editarietà multipla porti con se più svantaggi che vantaggi perché alla fine il codice risulta essere molto più complicato e difficile da correggere rispetto a programmi che non ne fanno uso; per questo motivo in Java e altri linguaggi esiste solo l'ereditarietà singola. Per cercare di ottenere un meccanismo simile all'ereditarietà multipla in Java esistono le interfacce cioè un insieme di metodi astratti che dovranno essere implementati nelle classi che le estendono. Non avendo variabili o metodi concreti anche se due interfacce avessero due metodi con stessa firma coincidenti, non succederebbe nulla di male perché, appunto, questi metodi sono vuoti. Purtroppo il sistema delle interfacce non favorisce il riutilizzo del codice perché non si possono ereditare dei metodi concreti da più classi. Per ovviare a questo problema, in Scala e altri linguaggi, sono presenti i tratti.
I tratti sono simili alle interfacce di Java ma con la differenza di poter contenere campi e metodi concreti. Un esempio di tratto è il seguente:

```
trait HaFoglie {

    println("ho le foglie")

    var colore: String = "verdi"
```

```
    def getColore = colore
}
```

La dichiarazione di un tratto è simile alla dichiarazione di una classe con la differenza di usare la parola chiave `trait`. Un tratto può essere ereditato da una classe o un altro tratto. Questo viene fatto con la parola chiave `extends`:

```
class Albero extends HaFoglie{

    println("sono un albero")
}
```

Per ereditare da più tratti si usa la parola chiave `with`:

```
trait HaRadici {

    println("ho le radici")
}

class Albero extends HaFoglie with HaRadici{

    println("sono un albero")
}
```

Quando estendiamo una classe con l'ereditarietà singola abbiamo la possibilità di sovrascrivere qualsiasi metodo della superclasse.

2.4 Gerarchia dei tipi

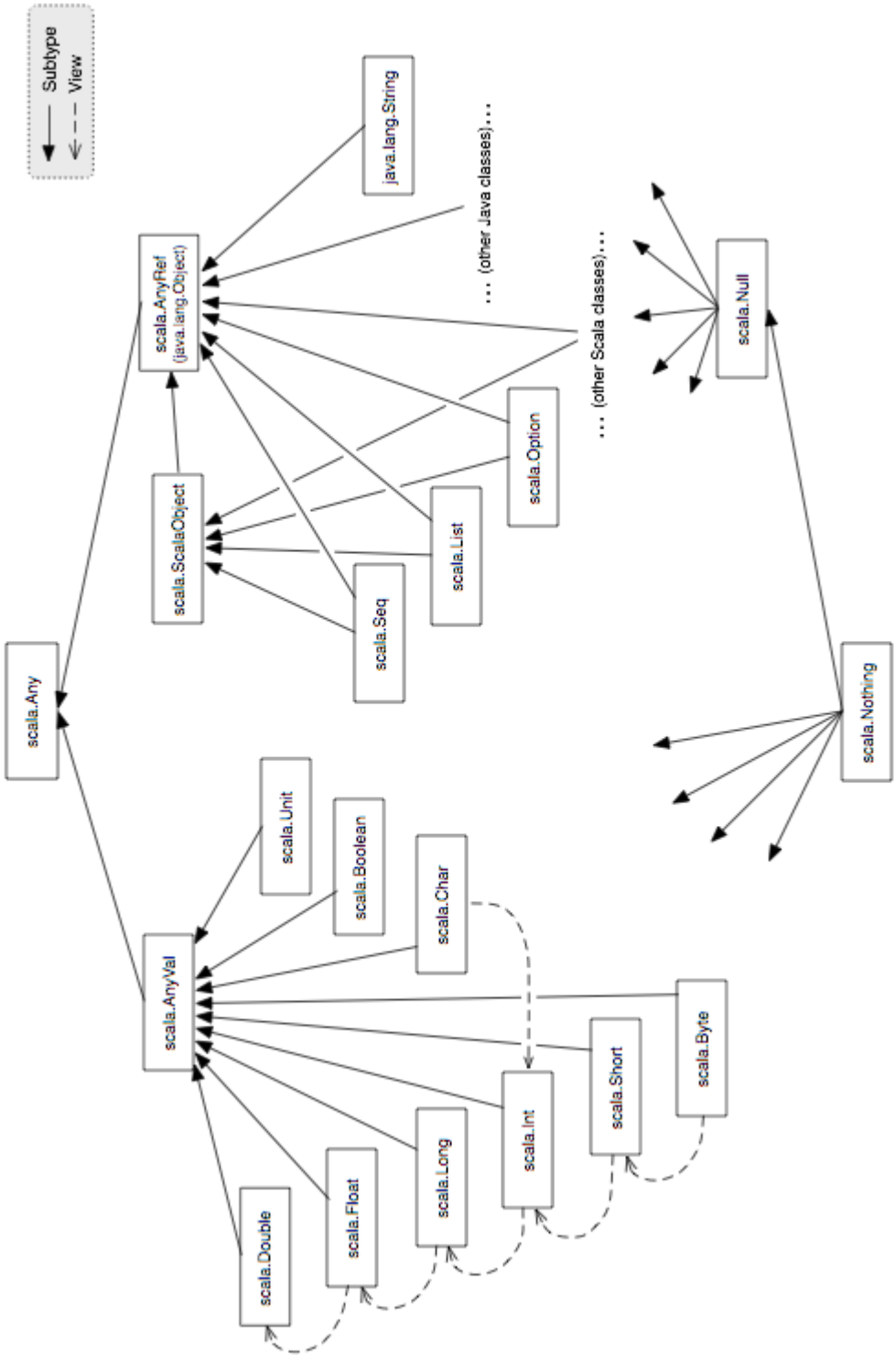
Come detto precedentemente in Scala ogni valore è un oggetto,esso sia numerico o una funzione, inoltre essendo un linguaggio basato su classi si ha che ogni valore è un'istanza di una classe.
Nella figura sottostante viene mostrata la gerarchia di classi in Scala. In alto possiamo vedere che la prima differenza è tra classi valore (che sono predefi-

nite e corrispondono ai tipi primitivi in Java) e classi referenziate. Da notare che oltre alle superclassi in cima, quali la radice `Any`, `AnyVal` ed `AnyRef`, abbiamo anche le sottoclassi: `Null` è la sottoclasse di tutti i tipi referenziati, mentre `Nothing` lo è per tutti i tipi.
La classe `Any` è la madre di tutte le classi in Scala, infatti è da lei che che ne derivano tutte le altre, essa definisce metodi final come ad esempio `!=`, `==`, `asIstanceOf[T]` (per la conversione di tipo).
Possiede due dirette sottoclassi che sono:

- `AnyVal`
  rappresenta la classe che contiene tutti i tipi valore, possiede tutte istanze di valore immutabile e nessun tipo può essere istanziato con la parola chiave `new`;

- `AnyRef`
  rappresenta la classe a cui appartengono tutti i tipi referenziati, ossia l'equivalente della classe `Object` di Java da cui derivano tutte le classi appartenenti a Java; un'altra sottoclasse importante di questo gruppo è `ScalaObject` da cui derivano tutte le classi fornite dalla piattaforma Scala come ad esempio `List`, `Seq`, `Option`.

La classe `Null` rappresenta il valore nullo solo per i tipi appartenenti a `AnyRef` mentre `Nothing` lo è per i tipi appartenenti a `AnyVal`. In realtà essendo `Nothing` una derivazione di tutte le classi esso può essere esteso anche ai tipi appartenenti alla classe `AnyVal`.


Subtype
View
java.lang.String
... (other Java classes)...
scala.AnyRef
(java.lang.Object)
... (other Scala classes)...
scala.Null
scala.Option
scala.ScalaObject
scala.List
scala.Seq
scala.Any
scala.Nothing
scala.Unit
scala.Boolean
scala.Char
scala.AnyVal
scala.Byte
scala.Short
scala.Int
scala.Long
scala.Float
scala.Double

Capitolo 3

Programmare in Scala su Android

3.1 Prerequisiti

Il primo passo per lo sviluppo di App Android in Scala sarà installare i seguenti kit:

- **JDK**[1]
  Android supporta tutte le caratteristiche di Java 7, ma solo un sottoinsieme di funzionalità di Java 8, che comunque variano a seconda della versione della piattaforma Android utilizzata.
  Per questo JDK 7 dovrebbe essere la nostra scelta, almeno per ora. Se si dispone di una versione più recente non c'è alcun problema, per far funzionare il tutto si deve specificare il target di compilazione nel file `built.sbt` settandolo alla versione 1.7:

```
javacOptions in Compile ++= "-source" :: "1.7" ::
   "-target" :: "1.7" :: Nil
```

[1] `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

- **Android SDK**[2]
  kit di sviluppo contenente gli strumenti, le risorse, gli emulatori e le piattaforme per rilasciare applicazioni Android.

3.2 SBT e Plugin Android SDK

Il prossimo passo necessario sarà decidere quale strumento di compilazione dovrebbe gestire le dipendenze, compilare e distribuire l'applicazione.
Tra tutti quelli esistenti, si è scelto SBT[3], ossia Simple Built Tool, che è lo strumento standard di compilazione Scala ed i suoi file di configurazione sono anche scritti in Scala. Ci sono anche altre alternative, come Ant, Maven o Gradle, ma a differenza di questi, SBT fornisce un ambiente di sviluppo più confortevole per i progetti Scala-based. Il sito ufficiale spiega come installarlo: `http:// www.scala-sbt.org/0.13/docs/Setup.html` .
Il plugin Android SDK[4] per SBT supporta tutte le configurazioni per progetti Android. Librerie di terze parti possono essere incluse manualmente, oppure possono essere aggiunte utilizzando la funzione `libraryDependencies` di SBT. Questo plugin è compatibile al 100% con il sistema di compilazione standard di Android. La maniera più semplice per installarlo è aggiungerlo globalmente, basta creare (se già non esiste) la cartella plugins nel seguente percorso:

```
$ HOME/.sbt/0.13/
```

ed aggiungervi il file `android.sbt` con all'interno questa linea:

[2] `https://developer.android.com/studio/index.html`
[3] `http://www.scala-sbt.org/`
[4] `https://github.com/scala-android/sbt-android`

```
addSbtPlugin("org.scala-android" % "sbt-android" % "1.7.5")
```

Qui di seguito sono riportati i comandi da eseguire:

```
$ cd
$ HOME/.sbt/0.13/plugins/
$ echo 'addSbtPlugin("org.scala-android" % "sbt-android" %
    "1.7.1")' > android.sbt
```

Alcuni comandi utili resi disponibili da questo plugin sono:

- `compile`
  compila tutti i sorgenti nel progetto, Java e Scala

- `android:package-release`
  costruisce un APK di pubblicazione firmato con una chiave, se configurata

- `android:package-debug`
  costruisce un APK di debug firmato con una chiave di debug

- `android:package`
  costruisce un APK per il progetto dell'ultimo tipo selezionato, con debug di default

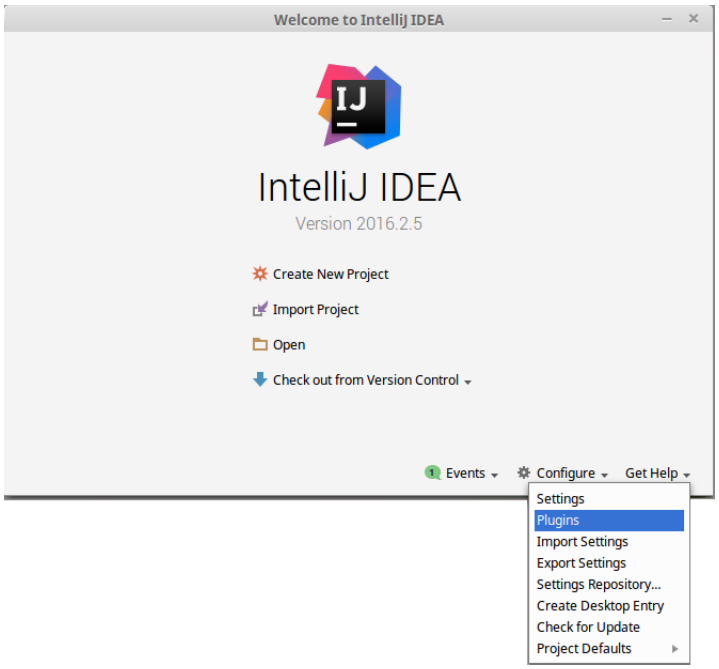
- `android:install`
  installa l'applicazione sul dispositivo

- `android:run`
  installa ed esegue l'applicazione sul dispsitivo

- `android:uninstall`
  disinstalla l'applicazione dal dispositivo

3.3 IntelliJ IDEA

L'IDE che meglio si presta allo sviluppo di questo progetto è IntelliJ IDEA, che è disponibile in due versioni: Ultimate e Community Edition. Quest'ultima è gratuita e fornisce le caratteristiche sufficienti di cui abbiamo bisogno per lo sviluppo di app Android in Scala.
Oltre all'IDE bisogna installare due plugin, per farlo è sufficiente avviare l'IDE e sulla finestra di benvenuto cliccare sulla voce `Plugin`:

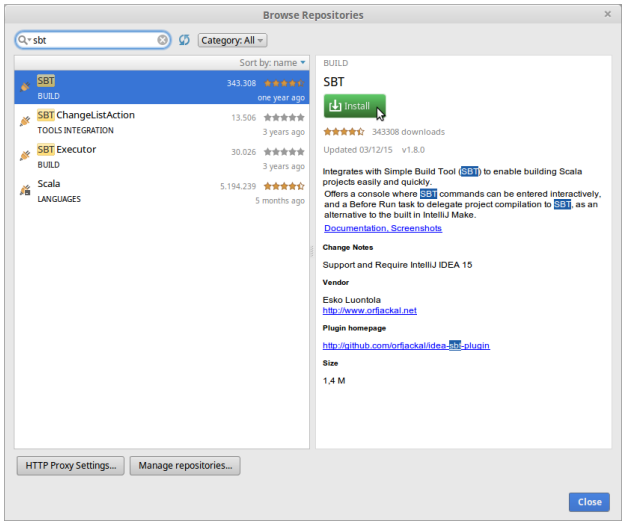


I plugin che ci serviranno sono:

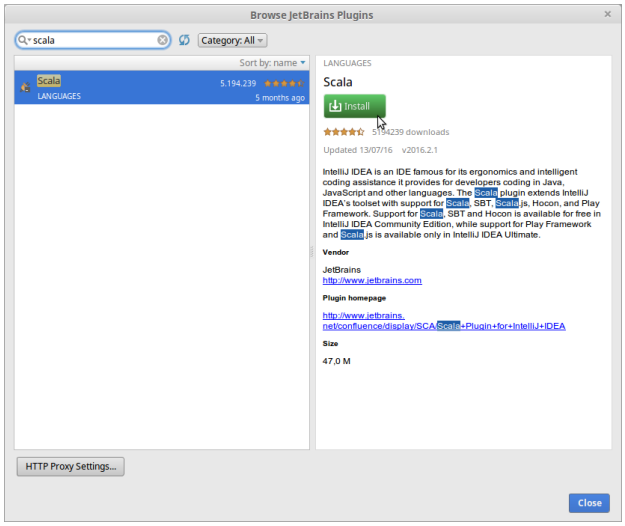
- **SBT**[5]
  il quale si integra con SBT per consentire facilmente e rapidamente la

[5] `http://github.com/orfjackal/idea-sbt-plugin`

costruzione di progetti Scala; inoltre offre una console in cui i comandi SBT possono essere inseriti in modo interattivo, ed una prima `Before Lunch` capace di delegare la compilazione del progetto ad SBT



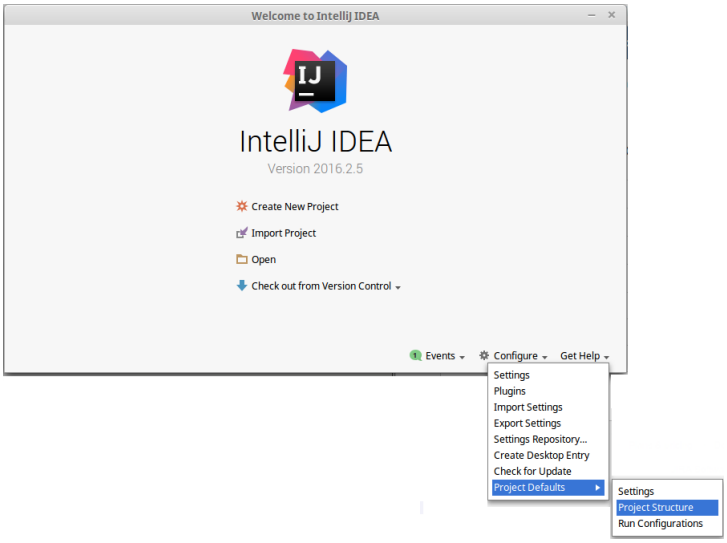
- **Scala**[6]
  necessario a creare ed eseguire applicazioni Scala

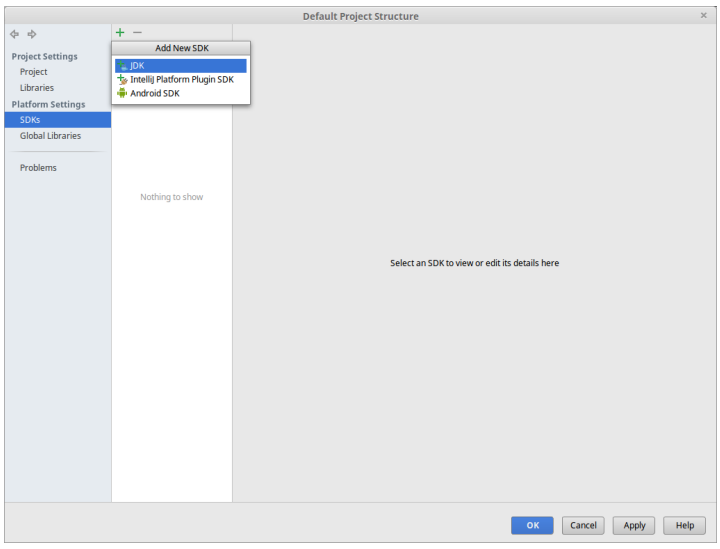


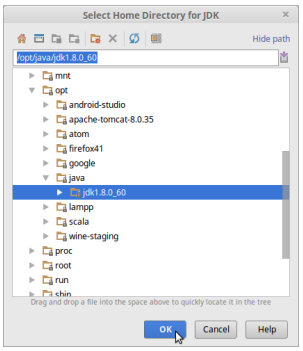
[6] `http://www.jetbrains.net/confluence/display/SCA/Scala+Plugin+for+IntelliJ+IDEA`

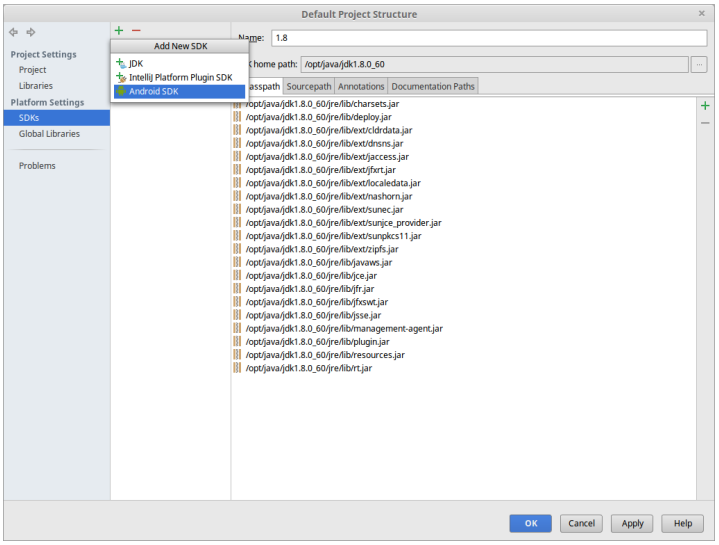
Ogni volta che installiamo un plugin, bisogna riavviare l'IDE per rendere effettiva la modifica.

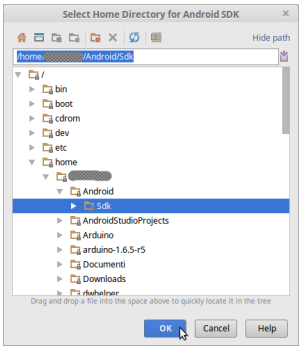
Il prossimo passo da fare sarà segnalare all'IDE i path che puntano alla posizione degli SDK che andremo ad usare, nel nostro caso JDK (da settare per primo) ed Android SDK. I passaggi per farlo sono illustrati di seguito:











3.4 La prima App

Iniziamo creando la cartella che conterrà il progetto ed il progetto stesso usando SBT ed il suo plugin, eseguendo i seguenti comandi:

```
$ mkdir myproject
```

```
$ cd myproject
$ sbt
sbt> gen-android <package_name> <project_name>
sbt> android:package
```

Il comando `gen-android <package_name> <project_name>` serve a creare il progetto e può essere usato anche nella versione `gen-android <platform_target> <package_name> <project_name>`, in quest'ultimo caso platform_target sta ad indicare con quale versione di Android si intende testare il programma; la versione si indica scrivendo per esempio `"android-23"`, altrimenti verrà impostata quella di default, che potrà essere modificata manualmente in un secondo momento nel file `build.sbt`

```
platformTarget := "android-23"
```

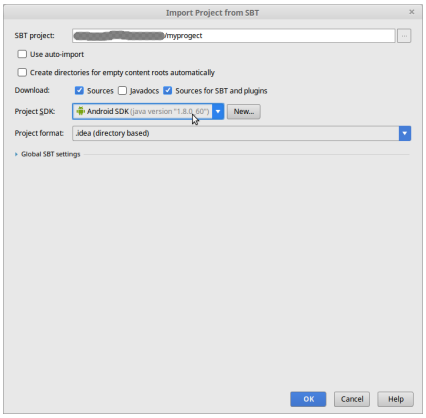
Il comando `android:package` ci serve a creare il file `.apk` che useremo successivamente.
Il progetto appena creato conterrà i file minimi necessari all'esecuzione dello stesso, e si presenterà come di seguito:

```
java.lang.IncompatibleCl
my-project/
|-- project/
|   |-- android.sbt
|   |-- build.properties
|-- src/
|   |-- androidTest/
|       |-- java/
|           |-- com/
|               |-- codurance/
|                   |-- Junit3MainActivityTest.java
|                   |-- Junit4MainActivityTest.java
|   |-- main/
|       |-- res/
```

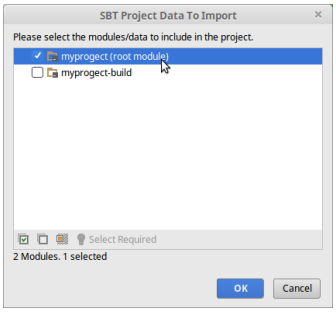
```
|           // Android resorces folders
|       |-- scala/
|           |-- package1/
|               |-- package2/
|                   |-- MainActivity.scala
|       |-- AndroidManifest.xml
|-- build.sbt
|-- lint.xml
```

Una volta aperto con IntelliJ basta seguire i seguenti passaggi per avviarlo:

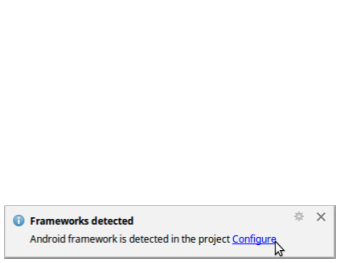
1. impostare `Android SDK` come `Progect SDK`

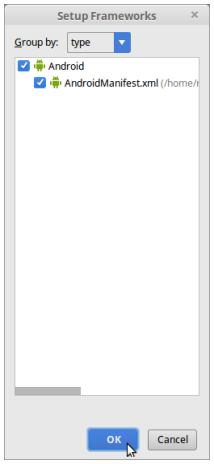


2. importare solo il `root module`

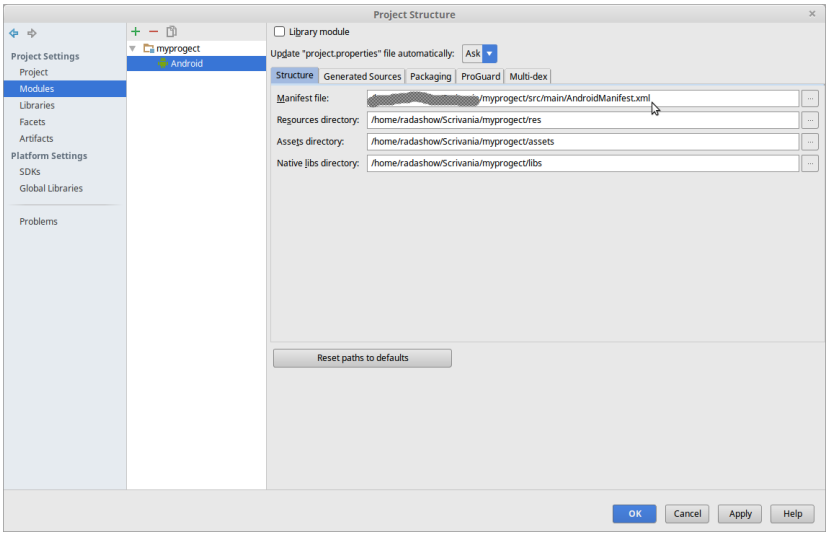


3. appena importato in progetto IntelliJ dovrebbe rilevare in automatico l'utilizzo del framework `Android` segnalandocelo tramite una messaggio, in tal caso dovremmo solo aggiungerlo al progetto



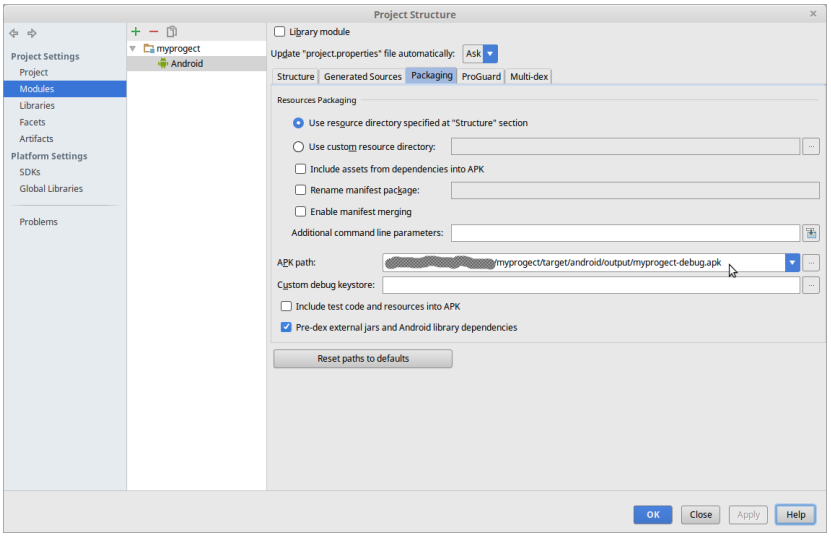


4. in `File > Project Structure > Modules` dovrebbe già esserci il framework `Android`, come visto al punto precedente, ma nel caso non ci fosse basta aggiungerlo andando sul simbolo `+`, dopo di che aggiornare il percorso del file `AndroidManifest.xml`

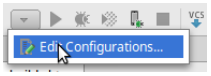


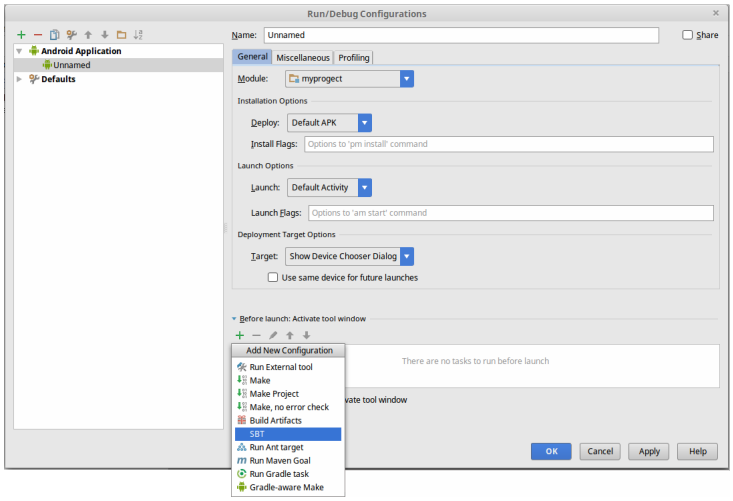
e sempre nella stessa schermata aggiungere il file `.apk`, che dovrebbe trovarrsi al seguente perxcorso: `/path CartellaProgetto/target/an-`

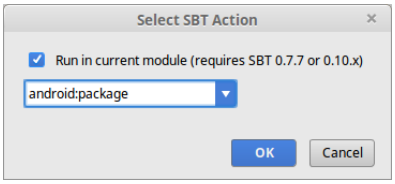
`droid/output/nomeapp(tutto minuscolo)-debug.apk`



5. riconfigurare il pulsante `Run` andando su `Edit configuration` e aggiungendo `Android Application` come configuration, poi nella sezione `Before launch` rimuovere `Gradle-aware Make` ed aggiungere il comando SBT necessario ad eseguire il progetto: `android:package`







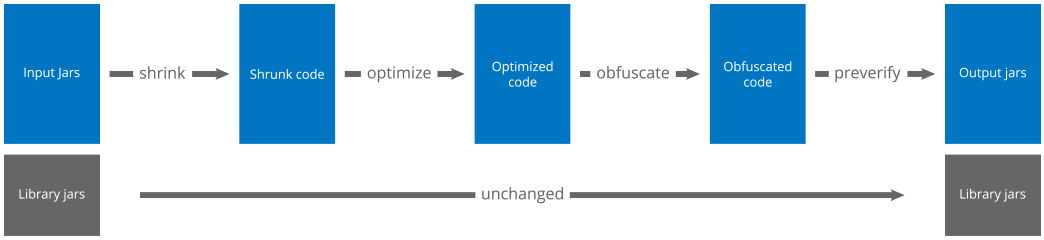
Arrivati a questo punto eseguendo il progetto su di un emulatore o un dispositivo reale dovrebbe già partire la prima app di esempio.

3.5 ProGuard

Lo strumento ProGuard[7] restringe, ottimizza e offusca il codice, eliminandone parti inutilizzate e rinominando le classi, i campi ed i metodi con i nomi sconosciuti. Il risultato è un file .apk di dimensioni più piccolo che risulterà più difficile da decodificare. ProGuard è l'ottimizzatore più importante per Java bytecode. Rende le applicazioni Java e Android fino al 90% più piccole e fino al 20% più veloc, fornendo anche una protezione minima contro il reverse engineering offuscando i nomi di classi, campi e metodi.

3.5.1 Fasi del processo

ProGuard legge i jar in ingresso e poi riduce, ottimizza, offusca, e pre-verifica; scrive poi i risultati di uno o più jar uscita. Nella figura sottostante[4] sono riportate le fasi del processo:



[7] https://www.guardsquare.com/en/proguard

1. **Shrinking**
   ProGuard determina in modo ricorsivo quali classi e quali membri delle classi vengono utilizzati, tutte le altre classi e membri vengono scartati, se l'applicazione viene distribuita con una libreria di terze parti, allora il numero di classi, campi e metodi che non vengono effettivamente utilizzati potrebbe essere molto significativo.

2. **Optimization**
   ProGuard ottimizza ulteriormente il codice, tra le altre ottimizzazioni, classi e metodi che non sono punti di ingresso possono essere resi privati, statici, o finali, accorpa più classi, rimuove gli attributi irrilevanti per l'esecuzione, ordina le variabili locali, raggruppa i metodi in un unica linea di codice, rimuove le costanti inutilizzate, ottimizza i costrutti if, for, switch e in generale rende il codice pičompatto e meno user-friendly.

3. **Obfuscation**
   ProGuard rinomina classi e membri della classe che non sono punti di ingresso, con nomi senza senso; mantenendo inalterati i punti di ingresso garantisce che essi possono ancora essere raggiunti con i loro nomi originali per non compromettere il corretto funzionamento del programma.

4. **Preverification**
   ProGuard esegue controlli su Java bytecode prima del runtime e annota file class a vantaggio della Java VM, questo è l'unico passo che non deve conoscere i punti di ingresso.

Ognuno di questi passaggi è facoltativo.

3.5.2 Comandi

Di seguito verranno elencati alcuni comandi che potrebbero essere utili nell'utilizzo di ProGuard, suddivisi per scopo.

Input/Output

- `-include [file name]`
  legge ricorsivamente le opzioni di configurazione dal nome del file dato;
- `-basedirectory [director yname]`
  specifica la directory di base per tutte i successivi file;
- `-injars [class path]`
  specifica i jar di ingresso (o aars, wars, ears, zips, apks, directories) dell'applicazione che dovranno essere processati;
- `-outjars [class path]`
  specifica i nomi dei jar in uscita (o aars, wars, ears, zips, apks, directories);
- `-skipnonpubliclibraryclasses`
  specifica di saltare le classi non pubbliche durante la lettura dei jar, per accelerare l'elaborazione e ridurre l'utilizzo della memoria di ProGuard; per impostazione predefinita, ProGuard legge classi di libreria non pubbliche e pubbliche allo stesso modo, tuttavia, le classi non pubbliche spesso non sono rilevanti, se non influenzano il codice effettivo del programma nei jar di ingresso; ignorandoli quindi si accelera ProGuard, senza influenzare l'uscita;
- `-dontskipnonpubliclibraryclasses`
  specifica di non ignorare le classi della libreria non pubbliche;
- `-target [version]`
  specifica il numero di versione da impostare nei file class elaborati, esso può essere uno di 1.0, 1.1, 1.2, 1.3, 1.4, 1.5 (o anche solo 5), 1.6 (o solo 6), 1.7 (o solo 7), o 1.8 (o solo 8); per impostazione predefinita, i numeri di versione dei file class vengono lasciate invariate (attenzione a non diminuire i numeri di versione dei file class, dato che il codice può contenere costrutti che non sono supportati nelle versioni precedenti).

Keep

- `-keep [class specification]`
  specifica classi metodi e campi da conservare come punto di ingresso al codice;
- `-keepclassmembers [classspecification]`
  specifica i membri della classe da conservare, se anche le classi sono conservate;
- `-keepclasseswithmembers [classspecification]`
- `-keep [classspecification]`
  specifica classi e membri delle classi da conservare, a condizione che tutti i membri delle classi siano presenti;
- `-keepnames [classspecification]`
  specifica classi e membri delle classi i cui nomi sono da preservare, se non vengono rimossi in fase di shrinking;
- `-printseeds [file name]`
  elencare in modo esaustivo le classi e membri delle classi appaiati con le varie opzioni -keep, l'elenco viene stampato sullo standard output o al file specificato; questo può essere utile per verificare se uno o più membri sono stati trovati trovati, soprattutto se si stanno utilizzando i caratteri jolly.

Shrinking

- `-dontshrink`
  specifica di non ridurre i file class in ingresso, di default questa opzione viene abilitata, quindi tutte le classi e membri delle classi vengono rimossi, ad eccezione di quelli elencati dalle varie opzioni -keep, e quelli da cui dipendono, direttamente o indirettamente;
- `-whyareyoukeeping`
  specifica di stampare i dettagli del motivo per cui le classi rilasciate e membri della classe vengono tenuti nella fase di shrinking, quest'informazione può essere utile per sapere il perché di qualche dato elemento presente in uscita.

Optimization

- `-dontoptimize`
  specifica di non ottimizzare i file class in ingresso, di default l'ottimizzazione è abilitata;
- `-optimizationpasses [n]`
  specifica il numero di ottimizzazione da eseguire, di default viene impostato ad uno; ovviamente più passaggi possono causare ulteriori miglioramenti;
- `-mergeinterfacesaggressively`
  specifica che le interfacce possono essere unite, anche se le classi che implementano quell'interfaccia non implementano tutti i metodi, ciò riduce la dimensione dell'output riducendo il numero totale di classi.

Obfuscation

- `-dontobfuscate`
  specifica di non offuscare i file class in ingresso, di default è attiva, classi e membri della classe ricevono nuovi nomi casuali brevi, ad eccezione di quelli elencati dalle varie opzioni -keep;
- `-printmapping [file name]`
  stampa la mappatura dai vecchi nomi ai nuovi nomi per le classi ed i membri della classe che sono stati rinominati,la mappatura viene stampato sullo standard output o al file specificato;
- `-obfuscationdictionary [file name]`
  specifica un file di testo da cui tutte le parole valide sono usate come i nomi dei campi e metodi offuscati;
- `-classobfuscationdictionary [file name]`
  specifica un file di testo da cui tutte le parole valide vengono utilizzate come nomi delle classi offuscate.
- `-dontusemixedcaseclassnames`
  specifica di non generare nomi di classi mischiando lettere maiuscole e minuscole, di defaulti nomi delle classi offuscate possono contenere un mix di caratteri maiuscoli e minuscoli;

- `-keeppackagenames [package]`
  specifica di non nascondere i nomi dei pacchetti dati;

Preverification

- `-dontpreverify`
  specifica di non preverificare i file class elaborati, di default i file di classe sono preverificati se sono destinati a Java Micro Edition o Java 6 o superiore;
- `-microedition`
  specifica che i file class elaborati sono destinati a Java Micro Edition.

General

- `-verbose`
  scrive qualche informazione in più durante la lavorazione;
- `-dontnote`
  specifica di non stampare le note circa i potenziali errori o omissioni nella configurazione;
- `-ignorewarnings`
  stampare eventuali avvisi su riferimenti non risolti e altri problemi importanti, ma continua l'elaborazione in ogni caso;
- `-printconfiguration [file name]`
  scrive l'intera configurazione utilizzata, inclusi file e variabili, la struttura è stampata sullo standard output o al file specificato;
- `-dump [nome del file]`
  scrive la struttura interna dei file class, dopo ogni trattamento, la struttura è stampata sullo standard output o al file specificato.

3.5.3 Attivazione in Android

Nel mondo Java, ProGuard è un passaggio facoltativo del processo di costruzione. Con Scala su Android diventa una necessità, in quanto è necessario

rimuovere classi e metodi non utilizzati perché la toolchain Android non riuscirà a compilare l' applicazione se contiene più di 65.536 tra metodi e/o campi (il cosiddetto limite di 65k). Questo limite è facile da raggiungere quando si ha la libreria Scala nelle dipendenze.
Per attivare ProGuard è sufficiente aggiungere le seguenti righe al file `built.sbt`:

```
// attiva ProGuard per Scala
proguardScala in Android := true

// attiva ProGuard per Android
useProguard in Android := true

// imposta opzioni per Proguard
proguardOptions in Android ++= Seq(
        "-ignorewarnings",
        "-keep class scala.Dynamic")
```

3.6 MultiDex

Molte volte l'utilizzo di ProGuard non è sufficiente per rientrare nel limite dei 65k.
Con Android 5.0 Runtime (ART) si è sostituita la macchina virtuale Dalvik (DVM), dotata di una modalità MultiDex, che permette ad uno sviluppatore di superare questo problema.
Ogni pacchetto APK ha almeno un file `.dex` ed è lì che è memorizzato tutto il codice eseguibile di un'applicazione. Questo include non solo la nostra implementazione, ma anche tutto il codice delle librerie a cui facciamo riferimento. Il file `.dex` ha un indice di indirizzamento a 16 bit, ciò sifnifica che 65536 rappresenta il numero totale di riferimenti che possono essere invocati dal codice all'interno di un singolo file. Se la nostra applicazione dovesse superare questo limite, lo strumento dx genererà un errore, causando il fallimento del confezionamento dell'APK. L'errore che verrà fuori è quello visibile nella

figura sottostante.

```
Error:Your app has more methods references than can fit in a single dex file.
See https://developer.android.com/tools/building/multidex.html
Error:Execution failed for task ':app:transformClassesWithDexForDebug'.
> com.android.build.api.transform.TransformException:
com.android.ide.common.process.ProcessException:
org.gradle.process.internal.ExecException: Process 'command '/Library
/Java/JavaVirtualMachines/jdk1.7.0_60.jdk/Contents/Home/bin/java'' finished with
non-zero exit value 2
```

Allora, che cosa dovremmo fare in questo caso? In realtà si potrebbe pensare come una grande occasione per rivedere il nostro codice. Infatti, prima di attivare il supporto multidex, dovrebbero essere fatti due passaggi. In primo luogo, rivedere le dipendenze usate nella nostra applicazione, al fine di escudere quelle superflue. La seconda opzione che abbiamo è quella di ridurre il codice utilizzando Proguard, come visto nel capitolo precedente. Se nessuna di queste due opzioni è sufficiente al nostro scopo, cioè produrre un file `.dex` con meno di 65536 metodi, la strada da percorrere è quella di consentire il supporto multidex.
Il file `.dex` è il cuore di ogni `.apk`, simile al file `.class` di Java, esso contiene il bytecode compilato dell' applicazione. Ma invece di creare un file `.class` per ogni classe (come fa javac), il file .dex contiene l'intero codice di byte dell' applicazione (incluse le dipendenze); e questo è, naturalmente, il file in questione del limite di 65k.
Ora, da Android 5.0 quel limite esiste ancora, ma si può dividere il bytecode in diversi file `.dex`, questo ci permetterà teoricamente di ignorare il limite.
Tutto ciò funziona anche per dispositivi Android con una versione precedente alla 5.0, perché Google fornisce una libreria di supporto per la macchina virtuale Dalvik.
Per abilitare il Multidex bisogna aggiungere la libreria:

```
"com.android.support" % "multidex" % "1.0.0"
```

e la righa seguente al file `built.sbt`:

```
dexMulti in Android := true
```

3.7 Akka

Uno degli obiettivi di questa tesi è l'integrazione di Scafi in Android. Scafi (di cui parleremo dopo) è un progetto composto da diversi moduli, ed uno di questi (`scafi-distributed`, che fornisce una piattaforma distribuita a supporto dell'esecuzione di sistemi aggregati) utilizza Akka, quindi si è provveduto prima a controllare che non ci fossero problemi tra quest'ultimo ed Android.
Ma cos'è Akka[8]? Akka è un toolkit gratuito ed open-source con lo scopo di semplificare la costruzione di applicazioni concorrenti e distribuite sulla JVM. Si integra nei progetti come una qualsiasi altra libreria. Dalla versione 2.10 in poi di Scala, è divenuta la piattaforma ad attori di riferimento. Il suo uso, per chi avesse intenzione di sviluppare applicazioni concorrenti (actor-based), è divenuto una necessità, poiché il package `scala.actors` è stato deprecato. Akka viene sviluppato per la forte necessità di avere applicazioni che siano, senza essere troppo complicate, correttamente distribuite, concorrenti, tolleranti agli errori e scalabili. Tramite il paradigma ad attori esso riesce a portare il livello di astrazione più in alto di quanto facesse la vecchia libreria ed a facilitare lo sviluppo di software scalabile e flessibile. Di seguito vi è elencato cosa offre questa piattaforma.

- **Modello ad attori**
  - astrazioni semplici e di alto livello per la distribuzione, la concorrenza e parallelismo;
  - modello di programmazione message-driven asincrono, non bloccante e altamente performante;
  - processi di tipo event-driven estremamente leggeri.
- **Fault Tolerance**
  si adotta la filosofia del "Let it crash", un modello che ha riscontrato

[8] `http://akka.io`

successo nell'industria delle telecomunicazioni la creazione di applicazioni self-heal e sistemi che non si bloccano mai.

- **Trasparenza dei percorsi**
  tutto è progettato per funzionare in un ambiente distribuito, perciò ogni interazioni fra attori è effettuata utilizzando passaggi di messaggi puri e tutto in modo asincrono.

- **Persistenza**
  cambiamenti di stato di un attore possono opzionalmente essere conservati e riprodotti quando l'attore viene avviato o riavviato, qQuesto permette agli attori di recuperare il loro stato, anche dopo un blocco della JVM o quando viene eseguita la migrazione a un altro nodo.

Per permettere ad Akka di funzionare su Android è necessario prima aggiuggere le librerie da usare (ovvio):

```
val akkaVersion = "2.3.7"
val akkaActor = "com.typesafe.akka" % "akka-actor_2.11" %
                      akkaVersion
val akkaRemote = "com.typesafe.akka" % "akka-remote_2.11" %
                    akkaVersion
libraryDependencies ++= Seq(akkaActor, akkaRemote)
```

Da notare che l'ultima versione di Akka (2.4.0), necessita di Java 8 per funzionare, che non è ancora completamente supportato da Android.
sarà poi fondamentale configurare correttamente ProGuard, in modo da non escludere nessuna classe indispensabile al corretto funzionamento di Akka. Su Internet ho trovato diverse configurazioni di ProGuard per Akka, ma alla fine, ho dovuto eseguire il debug manualmente, lanciavo il progetto con codice Akka incluso e una volta che il codice era stato eseguito, aspettavo per il crash con un'eccezione di runtime, che mi avrebbe detto che classe mancava, poi aggiornavo la mia configurazione di ProGuard e ripetevo fino a quando

non ho ottenuto più errori.

In basso viene riportata la configurazione finale ottenuta.

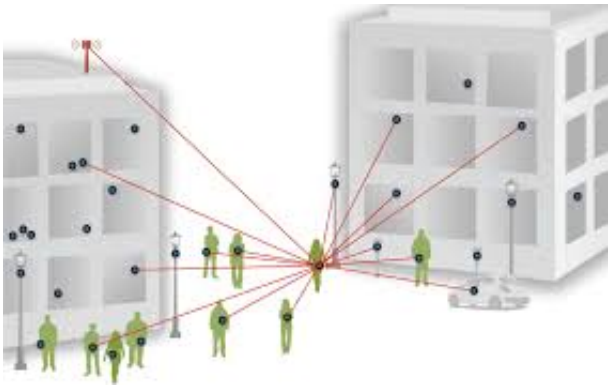
```
proguardOptions in Android ++= Seq(
    "-keep class akka.actor.Actor$class { *; }",
    "-keep class akka.actor.LightArrayRevolverScheduler { *; }",
    "-keep class akka.actor.LocalActorRefProvider { *; }",
    "-keep class akka.actor.CreatorFunctionConsumer { *; }",
    "-keep class akka.actor.TypedCreatorFunctionConsumer { *;
       }",
    "-keep class
       akka.dispatch.BoundedDequeBasedMessageQueueSemantics {
       *; }",
    "-keep class akka.dispatch.UnboundedMessageQueueSemantics {
       *; }",
    "-keep class
       akka.dispatch.UnboundedDequeBasedMessageQueueSemantics {
       *; }",
    "-keep class akka.dispatch.DequeBasedMessageQueueSemantics
       { *; }",
    "-keep class akka.dispatch.MultipleConsumerSemantics { *;
       }",
    "-keep class akka.actor.LocalActorRefProvider$Guardian { *;
       }",
    "-keep class
       akka.actor.LocalActorRefProvider$SystemGuardian { *; }",
    "-keep class akka.dispatch.UnboundedMailbox { *; }",
    "-keep class akka.actor.DefaultSupervisorStrategy { *; }",
    "-keep class macroid.akka.AkkaAndroidLogger { *; }",
    "-keep class akka.event.Logging$LogExt { *; }"
)
```

Capitolo 4

Utilizzare Scafi su Android

4.1 Aggregate Programming

La sempre crescente varietà di dispositivi collegati in rete (cellulari, orologi, automobili, e tutti i tipi di sensori) pervade i luoghi che frequentiamo tutti i giorni. Le interazioni tra di loro giocano un ruolo importante nella visione dell'Internet of Things (IoT).



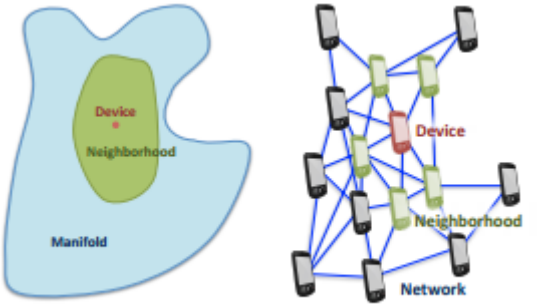
Tradizionalmente, l'unità di base di calcolo era il singolo dispositivo, le connessioni tra di loro svolgevano un ruolo secondario. Ma col passare del tempo questo aspetto ha preso sempre più importanza, facendo crescere la realizzazione di applicazioni distribuite in complessità. Questo ha causato problemi di progettazione, mancanza di modularità e riutilizzabilità, difficoltà nella

distribuzione e problemi di collaudo e manutenzione.
La programmazione aggregata fornisce un'alternativa che semplifica notevolmente la progettazione, realizzazione e manutenzione di complessi sistemi software, in particolare nel contesto dell'IoT, cyber-physical systems, pervasive computing, robotic swarms, ed in generale sistemi situati su larga scala. L'unità base di calcolo non è più il singolo dispositivo, ma piuttosto una raccolta cooperante di dispositivi: dettagli del loro comportamento, posizione e il numero sono astratti.
I tipici linguaggi di programmazione device-centric costringono un programmatore a concentrarsi su singoli dispositivi e le loro interazioni. Di conseguenza, diversi aspetti di un sistema distribuito di solito finiscono impigliati insieme: l'efficacia e l'affidabilità delle comunicazioni, il coordinamento di fronte a modifiche e guasti, e la composizione di comportamenti di dispositivi diversi da una parte all'altra di regioni diverse. ciò rende molto difficile progettare, eseguire il debug, mantenere, e comporre complesse applicazioni distribuite.
La programmazione aggregata tenta in generale di affrontare questo problema fornendo astrazioni componibili che separano questi aspetti:

- la comunicazione da dispositivo a dispositivo è realizzata interamente implicitamente, con astrazioni di livello superiore per ottenere un buon compromesso tra efficienza e robustezza;
- i metodi di coordinamento distribuiti sono incapsulati come operazioni aggregate-level (per esempio, misurando la distanza da una regione, diffondendo un valore da gossip, campionando una raccolta di sensori ad una certa risoluzione nello spazio e nel tempo);
- il sistema complessivo è specificato componendo operazioni di aggregate-level, e poi trasformato in un completo sistema distribuito grazie ad una mappatura adeguata.

Molte applicazioni di programmazione aggregata coinvolgono insiemi di dispositivi, dove operazioni geometriche e il flusso di informazioni forniscono

un'utile fonte di astrazione aggregate-level. Da questo, sono stati sviluppati diversi metodi, tutti basati sulla visualizzazione di un insieme di periferiche come un'approssimazione di un campo continuo (figura sottostante).



(a) Continuous Space (b) Discrete Network

L'approccio della programmazione aggregata è basato su 3 principi (tratti dall'articolo "Aggregate Programming for the Internet of Things"[6]):

- *the "machine" being programmed is a region of the computational environment whose specic details are abstracted away—perhaps even to a pure spatial continuum*;

- *the program is specified as manipulation of data constructs with spatial and temporal extent across that region*;

- *these manipulations are actually executed by the individual devices in the region, using resilient coordination mechanisms and proximity-based interactions.*

così facendo, questo paradigma nasconde i complicati meccanismi di coordinazione, in questo modo facilita la progettazione e favorisce la costruzione di sistemi più modulari: questo approccio risulta molto importante nell'emergente scenario dell'IoT, caratterizzato da un continuo aumento di applicazioni per sistemi di larga scala, che sfruttano interazioni opportunistiche e di prossimità.

4.2 Scafi

Scafi[1](acronimo di "SCAla FIelds") è un framework di programmazione aggregata per Scala, si compone di due parti principali:

- un DSL Scala-internal per esprimere i calcoli di aggregazione;
- una piattaforma distribuita che supporti la configurazione e l'esecuzione di sistemi aggregati.

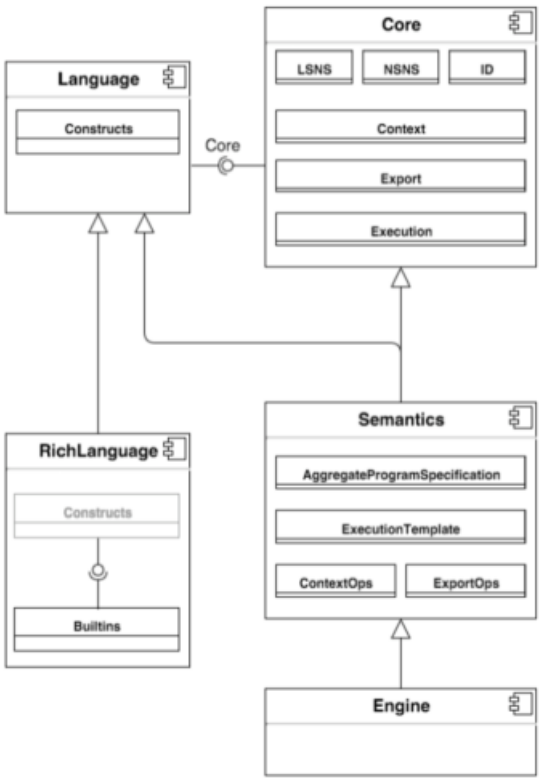
Un sistema aggregato è costituito da una vasta quantità di dispositivi di calcolo (chiamati anche nodi, in una rete di elementi; o punti, in uno spazio), che eseguono tutti lo stesso programma aggregato ripetutamente e in modo asincrono. In base alle informazioni accordate (ad es, valore dei sensori o dati del vicinato), diversi dispositivi possono assumere diversi rami della computazione. Lo stato dell'intero sistema, pertanto, può essere rappresentato come il campo di valori calcolati da ciascun dispositivo. L'interazione dipenderà dalla distanza tra dispositivi, cioè un dispositivo può comunicare direttamente solo con i suoi vicini. La comunicazione viene effettuata ripetutamente trasmettendo l'ultimo stato (chiamato `export`) all'intero vicinato. Il calcolo dello stato di ogni dispositivo è costituito dai seguenti passaggi:

1. creazione del contesto di esecuzione, che comprende l'ultimo valore calcolato, il più recente `export` ricevuto dai vicini, ed un'istantanea dei valori dei sensori locali;
2. esecuzione locale del programma complessivo, che produce il nuovo stato;
3. trasmissione all'intero vicinato dello stato in modalità broadcast;
4. attivazione degli attuatori, con in ingresso il risultato del calcolo.

Il progetto è stato realizzato dal professor Mirko Viroli ed esteso dall'Ing. Roberto Casadei. Scafi si basa sul paradigma di programmazione aggregata,

[1] `https://bitbucket.org/scafiteam/scafi`

con la differenza però che non si tratta di un vero e proprio DSL ma sono piuttosto un insieme di primitive Scala che data la loro compattezza e utilità possono essere considerate come un vero e proprio linguaggio di programmazione specifico per la programmazione aggregata.
La figura seguente ne mostra l'architettura e ne mette in evidenza i suoi componenti chiave.



Di seguito, senza entrare nel dettaglio, una breve descrizione dei suoi componenti cardine.

Core definisce le astrazioni base e gli elementi architetturali che saranno poi raffinati dai componenti figli;

Language basato sulle astrazioni definite in Core, definisce i principali Constructs del DSL, quest'ultimo elemento espone le primitive del fiedl calculus come metodi;

RichLanguage provvede a rendere il linguaggio più espressivo;

Semantics estende la parte sintattica e strutturale di Language, raffina le astrazioni di Core e fornisce una semantica per i Constructs;

Engine rende eseguibile il componente Semantics.

L'implementazione di questi componenti base costituenti l'architettura è effettuata in Scala. Le caratteristiche di questo linguaggio di programmazione permettono di creare famiglie di tipi ricorsivi che possono essere raffinati in modo incrementale.

4.3 Test su Android

Dopo aver verificato che i progetti Android possano essere sviluppati correttamente con Scala, si è potuto passare al test di Scafi. Questo era da considerarsi un requisito fondamentale, visto che Scafi è stato prodotto con la tecnologia messa a disposizione da Scala.
Per sperimenttare il funzionamento di questo framework sulla piattaforma Android si è effettuato un piccolo test costituito da:

- la creazione di un `object` che rappresenta il gradiente;
- una modifica nella classe `MainActivity.scala` al metodo `onCreate`.

Viste le ridotte dimensioni, se ne riporta direttamente il codice.

```
object MyProgram extends AggregateProgram {

    def gradient(source: Boolean): Double =
        rep(Double.MaxValue){
            distance => mux(source) { 0.0 } {
                foldhood(Double.MaxValue)((x,y)=>if (x<y) x else
                    y)
                    (nbr{distance}+nbrvar[Double](NBR_RANGE_NAME))
            }
```

```
        }

    def isSource = sense[Boolean]("source")

    override def main() = gradient(isSource)
}
```

```
class MainActivity extends AppCompatActivity {

    val net = simulatorFactory.gridLike(
        n = 10,
        m = 10,
        stepx = 1,
        stepy = 1,
        eps = 0.0,
        rng = 1.1)

    net.addSensor(name = "source", value = false)
    net.chgSensorValue(name = "source", ids = Set(3), value =
        true)

    var v = java.lang.System.currentTimeMillis()

    net.executeMany(
        node = MyProgram,
        size = 100000,
        action = (n,i) => {
            if (i % 1000 == 0) {
                println(net)
                val newv = java.lang.System.currentTimeMillis()
                println(newv-v)
                println(net.context(4))
                v=newv
```

```
            }
        })
    }
}
```

In questa simulazione messa a disposizione dal framework Scafi, abbiamo:

1. la primitiva `gridLike` che crea la rete dei nodi, a cui successivamente si impostano per ciascuno di essi i vari sensori con i relativi valori;
2. la primitiva `executeMany` avvia la simulazione.

Dai test effettuati su dispositivi reali risulta che Scafi viene eseguito correttamente, restituendo in output ciò che ci si aspetta. L'unico problema riscontrato si è avuto all'esecuzione dell'istruzione `println(net.context(4))`, dove il compilatore riportava il seguente errore:

```
java.lang.IncompatibleClassChangeError: The method
    'java.lang.String java.lang.Object.toString()' was expected
    to be of type interface but instead was found to be of type
    virtual (declaration of 'java.lang.reflect.ArtMethod'
    appears in /system/framework/core-libart.jar)
```

Si è notato, in seguito, che per risolverlo basta disabilitare il tool ProGuard e mantenere abilitato il Multidex. Questa soluzione comunque non è da ritenersi efficiente, bisogna insomma capire quali sono le classi da non escludere.

Conclusioni

Negli ultimi anni si stanno diffondendo sempre più i linguaggi che possono essere compilati in bytecode eseguibile su una JVM. Non tutti però possono essere utilizzati, chi per motivi di compatibilità, chi di prestazioni.
L'obiettivo che ci eravamo prefissati, prima di cominciare le ricerche, era capire se Scala, in futuro, potesse tornare utile anche per quanto riguarda il mondo della programmazione mobile.
Per raggiungere il nostro scopo, bisognava innanzitutto riuscire a configurare l'ambiente di sviluppo. Come IDE si è scelto IntelliJ IDEA, perché tra tutti era quello che, con i suoi plugin, si prestava meglio al nostro caso. Dopo un po' di ricerche on-line siamo riusciti nel nostro intento.
Indubbiamente, l'utilizzo di Scala, la cui sintassi è concisa, elegante e flessibile comporta molti vantaggi allo sviluppatore. Un problema a cui si va incontro quando si tenta si sviluppare app Android con un linguaggio diverso da Java, quello ufficiale e documentato, è l'incremento del tempo di compilazione. Questo è dovuto non solo alla grandezza delle libreria da includere, ma anche all'uso di Proguard e all'abilitazione del Multidex (di cui non possiamo farne a meno).
Tuttavia, il supporto a questi nuovi linguaggi di programmazione sulla piattaforma Android sta migliorando. Sicuramente a breve ci saranno nuovi sviluppi, e chi sa, magari un giorno uno di questi diventerà il nuovo linguaggio ufficiale Android.
Il secondo obiettivo, che aveva come prerequisito il successo del primo, consisteva nel testare la piattaforma Scafi su Android. Il test effettuato, nono-

stante l'errore riportato nel capitolo precedente, ha restituito gli output che ci si aspettava. Da qui si è giunti alla conclusione che non ci dovrebbero essere problemi nell'esecuzione del framework su dispositivi Android, anche se bisognerebbe fare altri test prima di dar per certo il risultato.

Bibliografia

[1] Bill Venners Martin Odersky, Lex Spoon. Programming in Scala. artima, ii edition, 2010.

[2] Gianluca Tartaggia. Il linguaggio di programmazione scala. Ingegneria Informatica Padova, 2012.

[3] Scala on Android. http://scala-on-android.taig.io/

[4] ProGuard Manual. https://www.guardsquare.com/en/proguard/manual/introduction

[5] Akka documentation. http://doc.akka.io/docs/akka/2.4/scala.html

[6] J. Beal, D. Pianini, and M. Viroli. Aggregate programming for the internet of things. IEEE Computer, 48(9):22–33, 2015.

[7] R. Casadei and M. Viroli. Towards Aggregate Programming in Scala. In First Workshop on Programming Models and Languages for Distributed Computing (PMLDC), 2016.

[8] Roberto Casadei. Aggregate Programming in Scala: a Core Library and Actor-Based Platform for Distributed Computational Fields. Scuola di Ingegneria e Archiettura, 2015.

[9] SAC2015-Protelis. http://web.mit.edu/jakebeal/www/Publications/SAC2015-Protelis.pdf

[10] J. Beal, M. Viroli, and F. Damiani. Towards a unified model of spatial computing. In 7th Spatial Computing Workshop, AAMAS 2014, Paris, France, May 2014.

[11] Simone Costanzi. Integrazione di piattaforme d'esecuzione e simulazione in una Toolchain Scala per aggregate programming. Scuola di Ingegneria e Archiettura, 2015.