Capitolo 1

Android e Java per Android

Come accennato nell'Introduzione, il primo capitolo è dedicato alla preparazione di tutto ciò che ci serve per sviluppare la nostra applicazione, che illustreremo nel dettaglio più avanti. Dedichiamo quindi queste prime pagine a una descrizione molto veloce di cosa sia Android e di quale sia la sua relazione con il linguaggio di programmazione Java, che sarà quindi l'argomento della seconda parte. Rispetto ai volumi precedenti ho deciso infatti di riprendere alcuni concetti della programmazione Java più volte richiesti dai lettori (solo quello che serve al nostro scopo). Ciò che daremo quasi per scontato saranno i concetti di programmazione a oggetti. Se il lettore si sente già a suo agio con il linguaggio potrà leggere solamente la prima parte e poi andare direttamente al prossimo capitolo. Per iniziare descriveremo la classica applicazione *Hello World*, che nasconde un numero elevatissimo di concetti i quali rappresentano spesso uno scoglio iniziale con il linguaggio. Vedremo poi che cosa è il *delegation model* e come si utilizza in ambiente Android e *mobile* in generale. A tale proposito parleremo di classi interne e generics. Concluderemo quindi con la trattazione di un costrutto introdotto in Java 5 che sta diventando molto importante specialmente se utilizzato insieme a strumenti di sviluppo e di build delle applicazioni, ovvero le Java Annotation.
Di solito nel primo capitolo si descrive quella che è l'operazione di installazione dell'ambiente di sviluppo, che questa volta abbiamo deciso di non trattare, in quanto in continua evoluzione; avremmo rischiato

In questo capitolo

- **Che cos'è Android**
- **I componenti principali di Android**
- **Introduzione a Java per Android**
- **Concetti object-oriented**
- **Il delegation model**
- **Le classi interne**
- **Generics**
- **Annotation**

di sprecare dello spazio per qualcosa che sarebbe diventato obsoleto entro breve tempo. Per questo rimandiamo alla documentazione ufficiale.

Che cos'è Android

È molto probabile che un lettore che ha acquistato questo testo sia già a conoscenza di cosa sia Android. Dedichiamo quindi queste poche righe a chiarire alcuni aspetti importanti. Innanzitutto Android non è un linguaggio di programmazione né un browser, ma un vero e proprio stack che comprende componenti che vanno dal sistema operativo fino a una virtual machine per l'esecuzione delle applicazioni. Caratteristica fondamentale di tutto ciò è l'utilizzo di tecnologie open source a partire dal sistema operativo, che è Linux con il kernel 2.6, fino alla specifica virtual machine che si è evoluta in questi anni, passando dall'utilizzo della Dalvik VM ad ART che è stata introdotta dalla versione 4.4 (Kitkat) e che, come vedremo, ha ottimizzato in modo evidente aspetti critici dal punto di vista delle performance come la gestione della memoria. Il tutto è guidato dall'Open Handset Alliance (OHA), un gruppo di una cinquantina di aziende (numero in continua crescita), il cui compito è quello di studiare un ambiente evoluto per la realizzazione di applicazioni mobili.

Architettura di Android

Per descrivere brevemente l'architettura di Android ci aiutiamo con la Figura 1.1, la quale ci permette di mettere in evidenza i componenti principali, organizzati secondo una struttura a *layer*, ovvero:

- Application;
- Application Framework;
- Android Runtime;
- Libraries;
- Linux Kernel.

Ricordiamo che un'architettura a *layer* (`https://en.wikipedia.org/wiki/Abstraction_layer`) permette di fare in modo che ciascuno strato utilizzi servizi dello strato sottostante per fornire allo strato superiore altri servizi, di più alto livello. Nel caso dell'architettura di Android lo strato di basso livello è rappresentato da un *kernel Linux* che contiene l'implementazione di una serie di *driver* di interazione con l'hardware, per l'utilizzo, per esempio, dello stack Bluetooth, della memoria, della batteria, ma anche di aspetti che vedremo essere fondamentali, come la gestione della sicurezza e la comunicazione tra processi (*Binder IPC*). Questo è il layer di competenza dei vari costruttori di dispositivi, che dovranno creare i driver per il proprio hardware, in modo da sfruttarne al massimo le caratteristiche e potenzialità. I servizi offerti dai driver contenuti nel Kernel Linux vengono quindi utilizzati da una serie di componenti che fanno parte del layer che abbiamo indicato come *Libraries*. Si tratta di componenti implementati per lo più in codice nativo, e quindi C/C++, ma che espongono delle interfacce Java per l'interazione con servizi classici di un dispositivo mobile, come quello della persistenza dei dati (*SQLite*), grafica (*OpenGL-ES*), gestione dei font (*FreeType*) e altro ancora.

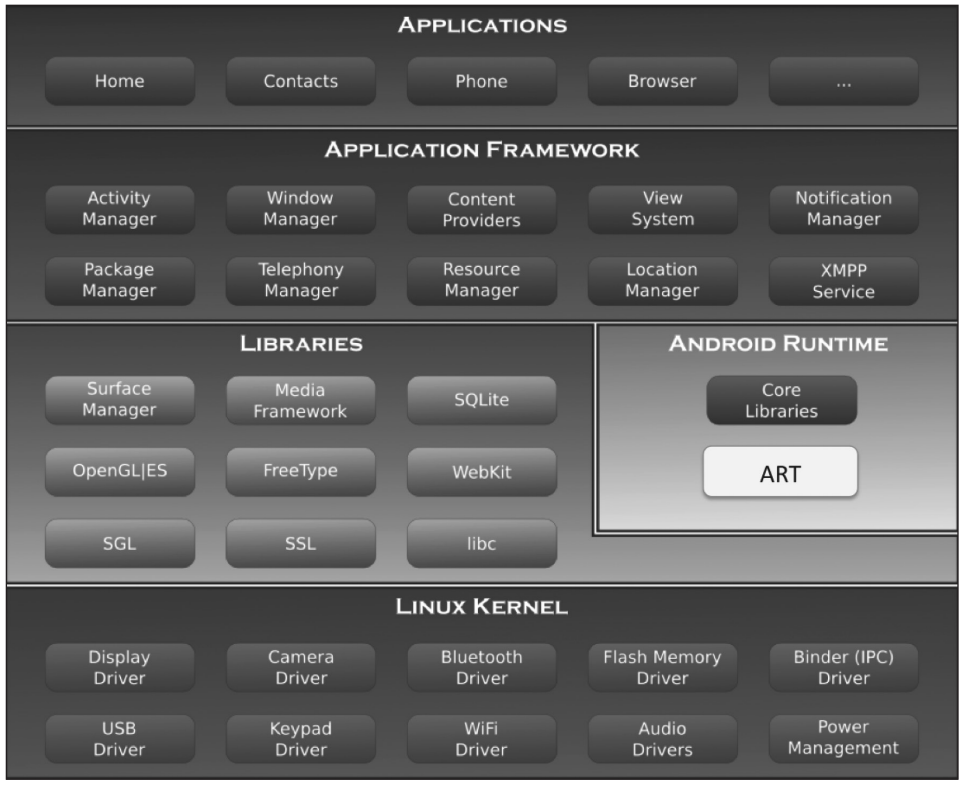

Figura 1.1 Architettura di Android (fonte: https://en.wikipedia.org/wiki/Android_(operating_system)).

Abbiamo accennato al fatto che si tratta di componenti implementati, per motivi di performance, in C/C++, ma che espongono delle interfacce Java, le quali vengono utilizzate da un altro componente fondamentale che si chiama *Core Libraries*. Si tratta di tutte le librerie che vedremo in questo testo e che ci permetteranno di creare le nostre applicazioni Android utilizzando Java come linguaggio di programmazione. In sintesi le nostre applicazioni utilizzeranno le API messe a disposizione dalle *Core Libraries* per accedere ai servizi implementati dal layer delle *Libraries.* Il tutto viene poi eseguito dal componente chiamato ART che rappresenta una novità introdotta in modo opzionale (in alternativa alla *Dalvik Virtual Machine*) a partire dalla versione 4.4 (KitKat) di Android e che è diventata l'unica disponibile dalla versione 5.0 (Lollipop). Il layer successivo è quello rappresentato dall'*Application Framework*, il quale utilizza servizi sia del *Runtime* sia delle *Libraries* e che contiene una serie di componenti di alto livello utili alla realizzazione di tutte le applicazioni Android. Si tratta di componenti che studieremo nel corso dei vari capitoli e che rappresentano i mattoni principali di tutte le applicazioni; sia quelle della piattaforma sia quelle che creeremo o scaricheremo dal *Market*, che, di fatto, compongono l'ultimo layer che si chiama *Application*.

Nella precedente descrizione abbiamo parlato di Java come linguaggio di programmazione per la realizzazione delle applicazioni Android, senza però sottolinearne l'importanza. In generale, quando si parla di Java non si parla infatti solamente di un linguaggio, ma

anche di una serie di strumenti e soprattutto di un'implementazione della JVM (Java Virtual Machine) le cui specifiche sono state rilasciate inizialmente da Sun Microsystem e successivamente da Oracle. Queste specifiche hanno fatto in modo che ne venissero realizzate diverse implementazioni per i vari sistemi operativi, in modo da mantenere la promessa del "Write Once, Run Everywhere". In questo modo è possibile compilare il codice Java in quello che si chiama *bytecode*, che poi viene eseguito nelle diverse implementazioni della JVM. In Android, però, non abbiamo alcuna JVM, ovvero alcuna virtual machine che implementi le stesse specifiche di Oracle. Se da un lato la ragione di ciò poteva essere di tipo economico, dall'altro si era alla ricerca di un ambiente molto più efficiente rispetto a quelle che erano state le implementazioni della JVM per dispositivi mobili, ovvero la JME (Java Micro Edition). Per questo motivo era stata creata la *Dalvik Virtual Machine* e ora l'ART (Android RunTime). Come abbiamo detto il linguaggio Java è un qualcosa in un certo senso distinto dal bytecode che viene eseguito dalla VM. Questo significa che è possibile creare lo stesso bytecode da linguaggi diversi (JavaFX, Clojure, Groovy e così via), ma anche utilizzare lo stesso linguaggio per creare binari diversi, come nel caso di Android. Come vedremo, l'ambiente Android ci permette, in fase di build, di compilare il codice Java in bytecode, per poi convertirlo in un altro tipo di codice intermedio che si chiama *dex*, che è quello che effettivamente viene eseguito dalle nuove VM in grado di sfruttare al meglio le caratteristiche dei vari hardware.

I componenti principali di Android

Come accennato in precedenza, Android è una piattaforma, all'interno della quale vengono eseguiti alcuni componenti che costituiscono i mattoncini con cui sono create tutte le applicazioni. Ma che cosa distingue un componente da un qualunque altro oggetto? La caratteristica principale di un componente è sicuramente quella della sua riutilizzabilità, ma anche il fatto di possedere un ciclo di vita che regola le interazioni con quello che si chiama container. Ciascuna applicazione Android sarà quindi costituita da uno o più dei seguenti componenti, che andremo a descrivere brevemente per poi approfondirli durante lo sviluppo della nostra applicazione;

- Activity;
- Intent e IntentFilter;
- Broadcast Intent Receiver;
- Service;
- ContentProvider.

Activity

Se prendiamo il nostro smartphone e avviamo una qualunque applicazione, notiamo come essa sia composta di schermate. Eseguendo, per esempio, l'applicazione di Gmail, notiamo come vi sia la schermata con l'elenco delle ultime mail, selezionando le quali andiamo a un'altra schermata con il relativo dettaglio. Un'ulteriore schermata è quella che utilizziamo per la scrittura e l'invio di una mail. In sintesi, ciascuna applicazione è costituita da schermate, che permettono non solo la visualizzazione delle informazioni, ma anche l'inserimento delle stesse. In un'applicazione Android ciascuna di queste

schermate è descritta da `Activity` che, come vedremo, non saranno altro che particolari specializzazioni dell'omonima classe. Ciascuna schermata definisce principalmente due aspetti: l'insieme degli elementi grafici e la modalità di interazione con essi. Ciascun elemento grafico verrà descritto da particolari specializzazioni della classe `View`, che vengono posizionate sullo schermo secondo determinate regole di layout. Come vedremo, queste regole potranno essere definite attraverso righe di codice (modo imperativo) oppure attraverso opportuni documenti XML di layout (modo dichiarativo), sfruttando alcuni strumenti messi a disposizione dell'IDE, che nel nostro caso è Android Studio. Una `Activity` avrà quindi la responsabilità di gestione dei componenti della UI (*User Interface*) e le interazioni con i servizi di gestione dei dati. Se pensiamo al celeberrimo pattern *Model View Controller* (MVC – `https://en.wikipedia.org/wiki/Model-view-controller`) potremmo assegnare alla `Activity` responsabilità di `Controller`.
Ciascuna applicazione sarà quindi costituita da una o più `Activity`, ciascuna delle quali verrà eseguita, se non specificato diversamente, dal proprio processo, all'interno di uno o più *task*. Il compito degli sviluppatori sarà quindi quello di creare le diverse `Activity` non solo in relazione alla loro UI, ma anche in base alle informazioni che esse si scambiano. In questo contesto, di fondamentale importanza è la gestione del ciclo di vita delle attività, attraverso opportuni metodi di *callback*. Si tratta di un aspetto importante di Android, a seguito della politica di gestione dei processi delegata in gran parte al sistema, che, in base alle necessità, può decidere di terminarne uno o più. In quel caso si dovranno adottare i giusti accorgimenti per non incorrere in una perdita di informazioni.

Intent e IntentFilter

Come abbiamo accennato, l'architettura di Android è ottimizzata in modo da permettere il migliore sfruttamento possibile delle risorse disponibili. Per raggiungere questo scopo si è pensato di "riciclare" quelle attività che svolgono operazioni comuni a più applicazioni. Pensiamo, per esempio, al caso di invio di una mail o di un SMS a un nostro contatto. Se ciascuna applicazione gestisse i contatti in modo custom si avrebbero degli svantaggi sia dal lato dell'utente sia da quello dello sviluppatore. L'utente si troverebbe davanti delle UI diverse per eseguire un'operazione che invece dovrebbe essere fatta sempre nello stesso modo: la selezione di un contatto. Lo sviluppatore si ritroverebbe invece a dover sviluppare una funzionalità che dovrebbe essere fornita dall'ambiente. Per questo motivo si è deciso di adottare il meccanismo degli `Intent`, che potremmo tradurre in "intenzioni". Attraverso un `Intent`, un'applicazione può dichiarare la volontà di compiere una particolare azione senza pensare a come questa verrà effettivamente eseguita. Nell'esempio precedente il corrispondente `Intent` potrebbe essere quello che dice "devo scegliere un contatto dalla rubrica". Ecco che l'applicazione che ha la necessità di scegliere un contatto dalla rubrica non dovrà implementare questa funzionalità da zero, ma dovrà semplicemente richiamarla attraverso il "lancio" del corrispondente `Intent`, a cui risponderà sicuramente (nel caso dei contatti) almeno l'implementazione fornita dall'ambiente Android. In precedenza abbiamo però parlato del fatto che le applicazioni fornite dall'ambiente sono scritte utilizzando gli stessi strumenti che andremo a utilizzare per sviluppare le nostre. Questo significa che vorremmo poter implementare un modo custom di eseguire un'operazione e quindi di soddisfare un determinato `Intent`. Serve quindi un meccanismo che permetta di dire al sistema che un particolare componente è in grado di soddisfare un particolare `Intent`. Per fare questo si utilizza un `IntentFilter`

il quale non è altro che un meccanismo per informare la piattaforma delle azioni che i nostri componenti sono in grado di soddisfare. Questo meccanismo non vale solo per i contatti, ma per un qualunque `Intent`. La gestione degli Intent e dei corrispondenti `IntentFilter` è parte del lavoro degli sviluppatori nel processo di definizione delle varie `Activity` e del flusso di navigazione. Come vedremo, questo meccanismo non è tipico delle sole `Activity`, ma rappresenta uno dei concetti fondamentali di tutta la piattaforma.

Broadcast Intent Receiver

Abbiamo appena visto come i concetti di `Intent` e `IntentFilter` siano fondamentali nella gestione dei componenti di ogni applicazione Android. Lo scenario relativo alla selezione di un contatto è però particolare, nel senso che si tratta di gestire un'azione che è avviata dall'utente che intende, per esempio, inviare un messaggio a un amico. Specialmente negli ultimi mesi, con l'introduzione di dispositivi *wear*, assume sempre maggiore importanza la possibilità di reagire a eventi che non sono avviati dall'utente, ma che sono scatenati da eventi esterni, come l'avvicinarsi a una particolare location, il raggiungimento di un particolare obiettivo nel numero dei passi fatti, il fatto che la batteria sia scarica, fino alla ricezione di un messaggio, di una mail o di un evento push. Esistono, insomma, degli eventi di diverso tipo, tra cui quelli di sistema, cui le varie applicazioni devono poter associare particolari azioni. Questi componenti vengono descritti dal concetto di *Broadcast Intent Receiver*, che sono in grado di attivarsi a seguito del lancio di un particolare `Intent` che si dice appunto di broadcast. Come vedremo, sono componenti che non sono dotati di UI e che vengono associati a un particolare insieme di `IntentFilter` corrispondenti ad altrettanti `Intent` di broadcast. Il loro compito è quello di attivarsi in corrispondenza di particolari eventi, raccogliendone le informazioni da utilizzare poi per l'esecuzione di operazioni più complesse, come la visualizzazione di una notifica, l'avvio di un servizio o il lancio di un'applicazione.

Service

In precedenza abbiamo visto come le `Activity` permettano la descrizione delle schermate di un'applicazione che si susseguono una dopo l'altra a seconda dello schema di navigazione. A tale proposito, supponiamo di avviare un'applicazione lanciando l'attività A1. Da questa, supponiamo di selezionare un'opzione che permette il lancio dell'attività A2, la quale è ora visibile nel display del nostro smartphone. In precedenza abbiamo solo accennato al fatto che le due attività facciano parte dello stesso task e di come siano organizzate secondo una struttura a stack; A1 sotto e A2 sopra. Nell'ottica di un'estrema ottimizzazione delle risorse, potrebbe succedere che A1 venga eliminata dal sistema, in modo da dedicare tutte le sue risorse ad A2 o ad altri componenti in esecuzione. Il sistema dovrà preoccuparsi anche di ripristinare A1 qualora l'utente vi ritornasse selezionando il pulsante di back dall'attività A2. L'aspetto fondamentale, in questo caso, è comunque relativo al fatto che l'attività A1 potrebbe essere terminata per un certo periodo di tempo. Questo significa che nel caso in cui avessimo avuto bisogno di mantenere in vita un determinato componente per la memorizzazione di alcune informazioni, A1 non sarebbe stato un posto ideale. Si ha quindi la necessità di un meccanismo che permetta di "mantenere in vita" il più possibile alcuni oggetti (o, come vedremo più avanti, thread) senza

correre il rischio che questi vengano eliminati al fine di una politica di ottimizzazione delle risorse. Questo è il motivo dell'esistenza di un altro componente fondamentale, che si chiama `Service`. Dedicheremo molto spazio a questo tipo di componenti perché di fondamentale importanza. Per il momento possiamo pensare ai `Service` come a un insieme di componenti in grado di garantire l'esecuzione di alcuni task in background, in modo indipendente da ciò che è eventualmente visualizzato nel display, e quindi da ciò con cui l'utente, in quel momento, sta interagendo.

ContentProvider

Un aspetto fondamentale di ciascuna applicazione è rappresentato dalla gestione dei dati, ovvero dalla possibilità di renderli persistenti. Come vedremo, Android ci mette a disposizione diversi strumenti che ci permettono di gestire le informazioni in modo privato. Questo significa che ciascuna applicazione gestisce i propri dati e non può accedere a quelli gestiti dalle altre. Come abbiamo visto nell'esempio della selezione di un contatto dalla rubrica, può succedere che le informazioni gestite da un processo debbano essere messe a disposizione degli altri e quindi delle altre applicazioni. L'applicazione di invio di un'email utilizza i dati gestiti dall'applicazione dei contatti. Questo deve avvenire in modo controllato e sicuro, attraverso interfacce predefinite che caratterizzano il `ContentProvider`. Possiamo pensare a un componente di questo tipo come a un oggetto che offre ai propri client un'interfaccia per l'esecuzione delle operazioni di CRUD (*Create*, *Retrieve*, *Update*, *Delete*) su un particolare insieme di entità. Chi ha esperienza in ambito JEE può pensare al `ContentProvider` come a una specie di *DAO* (*Data Access Object*), il quale fornisce un'interfaccia standard, ma che può essere implementato in modi diversi interagendo con una base dati, su file system, su cloud o semplicemente in memoria.

Introduzione a Java per Android

Come abbiamo detto all'inizio, il linguaggio principale con cui si sviluppano le applicazioni Android è Java, e per questo motivo abbiamo deciso di dedicare qualche pagina a quegli aspetti del linguaggio che ci saranno più utili. Il lettore già a conoscenza del linguaggio può tranquillamente passare al prossimo capitolo, dove inizieremo lo sviluppo dell'applicazione che abbiamo scelto come pretesto per lo studio della piattaforma.

> **NOTA**
>
> Gli esempi che descriveremo in questa fase verranno messi a disposizione come progetti per eclipse, ma possono essere importati in modo semplice anche in altri IDE, compreso Android Studio, che vedremo nel prossimo capitolo.

Iniziamo allora dalla celeberrima applicazione *Hello World*, leggermente modificata, di cui riportiamo il codice di seguito:

```
package uk.co.maxcarli.introjava;

import java.util.Date;
```

```
/**
* The classic HelloWorld application
*/
public class HelloWorld {

    /*
    * The message to print
    */
    private static final String MESSAGE = "Hello World!";

    /**
    * This is the main method for the application
    *
    * @param args
    * The array of arguments
    */
    public static void main(String[] args) {
        final Date now = new Date();
        // We print the message
        System.out.println(MESSAGE + " " + now);
    }

}
```

Si tratta di un'applicazione molto semplice, che contiene allo stesso tempo moltissimi concetti del linguaggio Java. Il primo è rappresentato dalla definizione del *package* cui la nostra classe appartiene, attraverso l'omonima parola chiave seguita da un identificativo composto da parole minuscole separate dal punto. Si tratta di un modo per raggruppare tra loro definizioni che fanno riferimento a concetti comuni. Per esempio, il *package* `java.net` contiene tutte le definizioni e gli strumenti relativi al networking, il *package* `java.io` gli strumenti di gestione delle operazioni di I/O e così via. Gli strumenti che la piattaforma eredita dall'ambiente Java sono contenuti all'interno di *package* che iniziamo per `java` o `javax`, mentre quelli caratteristici della piattaforma Android sono contenuti in *package* che iniziano per `android`; si tratta, in entrambi i casi, di *package* riservati, che quindi non possiamo utilizzare per le nostre definizioni.

> **NOTA**
>
> Il lettore avrà notato che non abbiamo ancora parlato di *classi*, ma di definizioni. Questo perché un *package* non contiene solamente la definizione di *classi*, ma anche di *interfacce*, *enum* e *annotation*. Per semplificare, di seguito faremo riferimento alle classi ricordandoci però che lo stesso varrà per gli altri tipi di costrutti.

Sebbene non si tratti di qualcosa di obbligatorio, è sempre bene che una classe appartenga a un proprio *package*, il quale definisce delle regole di visibilità che vedremo tra poco. Se non viene definita, si dice che la classe appartiene al *package* di default. Come possiamo vedere nel nostro esempio, il nome del package segue anche una convenzione che prevede che esso inizi con il dominio dell'azienda, ordinato inversamente. Nel caso in cui avessimo un'azienda il cui dominio è del tipo `www.miazienda.co.uk`, i vari package delle applicazioni che l'azienda vorrà produrre saranno del tipo:

```
uk.co.miazienda
```

È importante inoltre sottolineare come non esista alcuna relazione gerarchica tra un *package* di nome

```
nome1.nome2
```

e il package

```
nome1.nome2.nome3
```

Si tratta semplicemente di due *package* di nomi diversi senza alcuna relazione gerarchica. La gerarchia si avrà invece nelle corrispondenti cartelle che si formeranno in fase di compilazione. Ultima cosa riguarda il nome della classe, che nel nostro caso non è semplicemente `HelloWorld`, ma `uk.co.maxcarli.introjava.HelloWorld`. Il nome di una classe è sempre comprensivo del relativo *package*; questa regola è molto importante e ci permette di distinguere, per esempio, la classe `List` del *package* `java.awt` per la creazione di interfacce grafiche, dall'interfaccia `List` del *package* `java.util` per la gestione della famosa struttura dati. Abbiamo detto che ogni classe dovrebbe essere contenuta in un *package*, che impone anche delle regole di visibilità. Ogni classe è infatti visibile automaticamente alle classi del proprio package e vede sempre tutte le classi di un package particolare che si chiama `java.lang`. Il package `java.lang` è quello che contiene tutte le definizioni più importanti, come la classe `Object`, da cui derivano, in modo diretto o indiretto, tutte le classi Java, la classe `String`, tutti i wrapper (`Integer`, `Boolean`…) e così via. Qualora avessimo bisogno di utilizzare classi di altri package è possibile utilizzare la parola chiave `import`. Nel nostro esempio abbiamo utilizzato l'istruzione:

```
import java.util.Date;
```

per importare in modo esplicito la classe `Date` del *package* `java.util`. Nel caso in cui avessimo la necessità di importare altre classi dello stesso *package* potremmo eseguire l'`import` di ciascuna di esse oppure utilizzare la seguente notazione, che permette di importare (rendere visibili) tutte le definizioni del *package* `java.util`:

```
import java.util.*;
```

Anche qui due considerazioni fondamentali. La prima riguarda un fatto già accennato, ovvero che l'istruzione precedente non comprende l'import delle classi del package `java.util.concurrent` (un "sottopackage" del precedente), il quale dovrebbe essere esplicitato attraverso questa istruzione:

```
import java.util.concurrent.*;
```

La seconda riguarda il fatto che un'operazione di `import` non è assolutamente un'operazione di `include`, ovvero un qualcosa che carica del codice e lo incorpora all'interno dell'applicazione. Il numero di `import` non va a influire sulla dimensione della nostra applicazione, ma descrive un meccanismo che permette alla VM di andare a cercare il bytecode da eseguire tra un numero predefinito di location. Nel caso utilizzassimo en-

trambe le precedenti definizioni e utilizzassimo, per esempio, la classe `Lock`, il compilatore prima e l'interprete poi andrebbero a ricercarne la definizione all'interno del package predefinito `java.lang`, quindi nel package `java.util` e infine nel package `java.util.concurrent`. Ovviamente la ricerca termina non appena la definizione cercata viene trovata. Eccoci giunti alla creazione della classe di nome HelloWorld attraverso la seguente definizione:

```
public class HelloWorld {

    - - -
}
```

La prima parola chiave, `public`, si chiama *modificatore di visibilità* e permette di dare indicazioni su dove la nostra classe può essere utilizzata. Java prevede quattro livelli di visibilità ma tre differenti modificatori, i quali possono essere applicati (con alcune limitazioni) a una definizione, ai suoi attributi e ai suoi metodi. I livelli di visibilità con i relativi qualificatori sono i seguenti:

- pubblico (`public`)
- friendly o `package` (nessun modificatore)
- protetto (`protected`)
- privato (`private`)

Nel nostro esempio la classe `HelloWorld` è stata definita pubblica e questo comporta che la stessa sia visibile in un qualunque altro punto dell'applicazione. Attenzione: questo non significa che se ne possa creare un'istanza, ma che è possibile almeno creare un riferimento di tipo `HelloWorld`:

```
HelloWorld hello;
```

Di seguito abbiamo utilizzato la parola chiave `class`, che ci permette di definire una classe, che sappiamo essere un modo per descrivere, in termini di attributi e comportamento, un insieme di oggetti che si dicono sue istanze. Abbiamo poi il nome della classe, che le convenzioni impongono inizi sempre con una maiuscola e quindi seguano la *camel notation*, che prevede di mettere in maiuscolo anche le iniziali delle eventuali parole successive.

> **NOTA**
>
> A proposito del nome della classe, esiste un'importante regola spesso trascurata. All'interno di un file sorgente di Java vi può essere la definizione di un numero qualunque di classi, interfacce e così via. L'importante è che di queste ve ne sia al massimo una pubblica, che dovrà avere lo stesso nome del file. Questo significa che all'interno di un file sorgente potremo definire un numero qualunque di classi di visibilità *package* e dare al file un nome qualsiasi. Se però solamente una di queste classi fosse pubblica (nel qual caso sarebbe comunque la sola), il file dovrà avere il suo stesso nome.

Il corpo della classe viene poi descritto all'interno di quello che si chiama blocco e che è compreso tra le due parentesi graffe {}. Lo vedremo anche in un paragrafo successivo, ma la definizione precedente è equivalente alla seguente:

```
public class HelloWorld extends Object {

    - - -

}
```

Ogni classe Java estende sempre un'altra classe che, se non definita in modo esplicito, viene impostata dal compilatore. Quella definita è una classe esterna o top level.Vedremo successivamente come si possono definire classi interne e anonime.
Per le definizioni top level, `public` è l'unico modificatore che possiamo utilizzare, in quanto la visibilità alternativa è quella `package` o `friendly`, che non utilizza alcun modificatore. In questo caso la classe sarebbe stata definita in questo modo:

```
class HelloWorld {

    - - -

}
```

La visibilità sarebbe stata ristretta solamente alle classi dello stesso package. È importante sottolineare come le classi top level non possano assolutamente avere visibilità `protected` o `private`.
Procedendo con la descrizione della nostra applicazione, notiamo la seguente definizione, che contiene ancora dei modificatori interessanti:

```
private static final String MESSAGE = "Hello World!";
```

Il primo elemento è il modificatore di visibilità `private`, il quale permette di limitare al solo file la definizione della costante `MESSAGE`.

> **NOTA**
>
> Qui abbiamo parlato di file e non di classe. Tale definizione, sebbene privata, sarebbe perfettamente visibile all'interno di eventuali classi interne.

Segue quindi la parola chiave `static`, che assume diversi significati a seconda dell'elemento cui è applicata. Nel caso degli attributi, il significato è quello di un valore condiviso tra tutte le istanze della corrispondente classe. Nel caso specifico, se creassimo un numero qualunque di istanze di `HelloWorld`, tutte condividerebbero lo stesso valore di `MESSAGE`. L'aspetto interessante è però relativo al fatto che un attributo `static` esiste anche se non esistono istanze della relativa classe. Per questo motivo si parla di attributi di classe, in quanto vengono creati nel momento in cui il bytecode viene letto e interpretato dal `ClassLoader`. Pertanto essi possono essere referenziati non attraverso un riferimento a un'istanza ma attraverso il nome della classe. Trascurando per un attimo il fatto che si tratti di un attributo privato, possiamo accedervi attraverso questa sintassi:

```
<Nome Classe>.<attributo static>

HelloWorld.MESSAGE
```

Quando il modificatore `static` è applicato a un metodo, si dice che è un modificatore di classe e quindi non collegato in alcun modo allo stato di una particolare istanza. Si tratta sostanzialmente di metodi di utilità che utilizzano solamente le informazioni passate attraverso i propri parametri oppure utilizzano altri membri statici. Vedremo tra poco cosa comporterà questa osservazione con il metodo `main()`. Come vedremo successivamente, il modificatore `static` può essere utilizzato anche per la definizione di un tipo particolare di classe interna, detta appunto statica. Il modificatore `static` non può infine essere utilizzato nella definizione di una classe che abbiamo definito *top class*.
Anche il terzo modificatore, `final`, è molto importante. Può essere applicato sia alle classi sia agli attributi e metodi, anche qui con significati diversi. Una classe `final` è una classe che non può essere estesa; il compilatore utilizza questa informazione per applicare tutte le possibili ottimizzazioni, in quanto è sicuro che alcune funzionalità non verranno modificate attraverso overriding. Nel caso di un metodo, il significato è quello di impedirne l'overriding, mentre nel caso di un attributo il significato è quello di definizione di una costante. Per essere più precisi, un attributo `final` è un attributo che può ricevere un valore solamente una volta e prima che venga invocato il costruttore della corrispondente classe.
Quella nel nostro codice è la definizione di una costante di nome `MESSAGE`, visibile solamente all'interno del file `HelloWorld`.
Talvolta è facile confondere il concetto di `private` e di `static` e ci si chiede come faccia qualcosa che viene condiviso tra tutte le istanze di una classe a non essere pubblico. In realtà sono concetti indipendenti tra loro. Il primo riguarda il fatto che si tratta di qualcosa che è visibile solamente all'interno della classe. Il secondo esprime il fatto che si tratta comunque di qualcosa che è associato alla classe e non alle singole istanze.
Passiamo finalmente a descrivere il metodo che caratterizza un'applicazione Java e che ne permette l'esecuzione:

```
public static void main(String[] args)
```

In relazione a quanto detto, notiamo che si tratta di un metodo pubblico e statico di nome `main` con un parametro di tipo array di `String`. È importante sottolineare come la presenza di un metodo di questo tipo sia necessaria alla definizione di un'applicazione Java. Una sola differenza in uno dei modificatori, nel nome o nel numero o tipo di parametri renderebbe la nostra classe non avviabile, in quanto l'interprete non troverebbe il metodo per la sua esecuzione.
Per eseguire la nostra applicazione da riga di comando dovremmo utilizzare la prossima istruzione, dove notiamo che la classe è specificata in modo comprensivo del relativo package:

```
java uk.co.maxcarli.introjava.HelloWorld
```

Il comando `java` non è altro che l'interprete, che caricherà il bytecode della nostra classe per eseguirlo. Ma come fa l'interprete a eseguire del codice che non conosce? C'è la necessità di un punto di ingresso standard che ogni applicazione Java ha e che l'interprete si aspetta di chiamare in fase di avvio. Questo è il motivo per cui il metodo deve essere pubblico, deve avere `void` come tipo restituito e si deve chiamare `main` con un array di `String` come tipo del parametro di ingresso, che rappresenta gli eventuali parametri passati. Anche qui la parte interessante sta nell'utilizzo della parola chiave `static`. L'interprete caricherà

il bytecode della classe specificata per cui tutti i membri statici saranno già definiti. Di seguito invocherà quindi il metodo `main` senza dover creare alcuna istanza di `HelloWorld`. Infine, all'interno del metodo `main()`, non facciamo altro che inizializzare un'altra variabile `final` da utilizzare per la visualizzazione del nostro messaggio.
È buona norma utilizzare il modificatore `final` il più possibile, rimuovendolo solamente nel caso in cui la corrispondente proprietà abbia la necessità di cambiare. Qui è importante fare una considerazione relativamente al concetto di costante e di immutabilità. Per intenderci, la seguente istruzione:

```
final MiaClasse mc = new MiaClasse();

mc.setProp(newalue);
```

è perfettamente valida, in quanto non viene cambiato il valore del riferimento `mc` che è stato definito come `final` ma, attraverso il metodo `setProp()`, ne viene modificato lo stato. Se questo non fosse possibile si parlerebbe di immutabilità.
Concludiamo allora con la seguente istruzione, che nasconde diversi concetti della programmazione Java:

```
System.out.println(MESSAGE + " " + now);
```

Da quanto visto, `System` è una classe che, non essendo stata importata, intuiamo appartenere al package `java.lang`. È una classe anche perché inizia con una lettera minuscola. Da questo capiamo anche che `out` è una proprietà statica sulla quale possiamo invocare i metodi `println()`, nel cui parametro notiamo una concatenazione di `String` attraverso l'operatore `+`. In Java non esiste la possibilità di ridefinire gli operatori, ma nel caso delle `String` il `+` ha il significato di concatenazione. Eseguendo l'applicazione si ottiene però un risultato di questo tipo:

```
Hello World! Thu Jun 06 23:40:45 BST 2015
```

La concatenazione di un oggetto di tipo `Date` a una `String` provoca il risultato di cui sopra. In realtà, quando un riferimento a un oggetto viene concatenato a un qualunque oggetto si ha l'invocazione del corrispondente metodo `toString()`, che tutti gli oggetti ereditano dalla classe `Object`.
Fin qui abbiamo visto le parole chiave principali e come si utilizzano all'interno della nostra classe. Il linguaggio mette a disposizione anche altri tipi di modificatori che avremo comunque occasione di incontrare durante lo sviluppo del nostro progetto.

Concetti object-oriented

Prima di proseguire con gli strumenti e i pattern più utilizzati in Android, facciamo un breve riassunto di quelli che sono i concetti di programmazione a oggetti più importanti:

- incapsulamento;
- ereditarietà;
- polimorfismo.

Sono concetti tra loro legati e sono stati introdotti con lo scopo di diminuire l'impatto delle modifiche nel codice di un'applicazione. Sappiamo, infatti, che il costo di una modifica cresce in modo esponenziale a mano a mano che si passa da una fase di analisi alla fase di progettazione e sviluppo. In realtà, il nemico vero e proprio è rappresentato dal concetto di *dipendenza*. Diciamo che un componente B dipende dal componente A se, quando A cambia, anche B deve subire delle modifiche. Da questa definizione capiamo come limitando la dipendenza tra i diversi componenti diminuisce il lavoro da fare. Un primo passo verso questo obiettivo è rappresentato dall'incapsulamento, secondo cui lo stato di un oggetto e il modo in cui esso assolve alle proprie responsabilità debbano rimanere nascosti e quindi non visibili agli oggetti con cui esso interagisce. Questo significa che i diversi oggetti devono comunicare invocando delle operazioni il cui insieme ne rappresenta l'interfaccia. Per comprendere questo concetto vediamo un semplice esempio con la classe `Point2D`, che contiene le coordinate di un punto in un piano. Una prima implementazione di questa classe potrebbe essere la seguente:

```
/**
* Point with 2 dimensions
*/
public class Point2D {

    public double x;

    public double y;

}
```

la quale è caratterizzata dall'avere due attributi con visibilità `public`. Questo significa che è possibile utilizzare la classe `Point2D` come segue:

```
Point2D p = new Point2D();
p.x = 10.0;
p.y = 20.2;
```

In queste poche righe di codice non sembra esserci nulla di male, anche se si potrebbero fare moltissime considerazioni dal punto di vista del *multithreading*. Il nostro punto ha comunque due attributi che rappresentano le coordinate `x` e `y`. Tutte le classi che utilizzano questo oggetto avranno delle istruzioni che usano, attraverso un riferimento di tipo `Point2D`, tali proprietà. Ma cosa succede se, in futuro, il punto del piano non fosse più rappresentato da `x` e `y`, ma da `r` e θ, che rappresentano rispettivamente il modulo e l'angolo del punto secondo un sistema di riferimento polare? Tutte le classi che utilizzano il codice precedente dovrebbero modificare il loro, non solo per il nome delle variabili, ma soprattutto dovrebbero provvedere alla conversione tra i due sistemi. In questo caso un buon incapsulamento ci avrebbe aiutato, in quanto ci avrebbe permesso inizialmente di creare questa classe (nei sorgenti forniti le classi hanno nomi diversi, in modo da farli convivere nello stesso progetto):

```
/**
* Point with 2 dimensions
*/
```

```
public class Point2D {

    private double x;

    private double y;

    public Point2D( double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public void setX( double x) {
        this.x = x;
    }

    public double getY() {
        return y;
    }

    public void setY( double y) {
        this.y = y;
    }

}
```

In questo caso gli altri oggetti interagiscono con il nostro punto utilizzando metodi del tipo `getXXX()` e `setXXX()`. Ma cosa succederebbe nel caso di modifica della modalità di memorizzazione dei dati da cartesiana a polare come nel caso precedente? In questo caso sarà sufficiente modificare la classe nel seguente modo:

```
public class Point2DB {

    private double r;

    private double theta;

    public Point2DB(double x, double y) {
        this.r = Math.sqrt(x * x + y * y);
        this.theta = Math.atan2(y, x);
    }

    public double getX() {
        return r * Math.cos(theta);
    }

    public void setX(double x) {
        final double y = getY();
        this.r = Math.sqrt(x * x + y * y);
```

```
        this.theta = Math.atan2(y, x);
    }

    public double getY() {
        return r * Math.sin(theta);
    }

    public void setY(double y) {
        final double x = getX();
        this.r = Math.sqrt(x * x + y * y);
        this.theta = Math.atan2(y, x);
    }
}
```

Indipendentemente dalla complessità dell'implementazione, gli utilizzatori non si accorgerebbero della differenza, in quanto la loro modalità di interazione sarà sempre del tipo:

```
Point2D p = new Point2D(10.0, 20.5);
double x = p.getX();
double y = p.getY();
```

ovvero attraverso la sua interfaccia, che come possiamo vedere non è cambiata. La lezione in tutto questo è comunque quella di nascondere il più possibile il modo in cui un oggetto esegue le proprie operazioni, così che gli eventuali utilizzatori non dipendano da esso. L'esempio appena descritto utilizza un particolare tipo di classe che spesso viene chiamata, in ambienti magari non mobile, entità o *POJO* (*Plain Old Java Object*). Si tratta, infatti, di una classe che non descrive operazioni, ma che permette la semplice memorizzazione di informazioni. In ambiente Android non è raro trovare delle classi descritte come nel primo caso, ovvero attraverso un solo elenco di attributi pubblici, senza l'utilizzo di un lungo elenco di metodi `get` (*accessor*) e `set` (*mutator*). Un caso in cui questo può avvenire è, per esempio, quello di classi interne la cui visibilità è garantita dalla classe esterna oppure classi a cui si accede da più thread. La regola generale prevede quindi di non scrivere del codice che già in partenza sappiamo essere inutile.
Diverso è il concetto di incapsulamento applicato a classi che hanno responsabilità che possiamo chiamare "di servizio". Anche qui facciamo un esempio relativo a una semplice classe che dispone di un metodo che permette di moltiplicare due interi. Una prima implementazione potrebbe essere la seguente, la quale prevede un metodo `multiply()` che moltiplica due valori interi sommando il secondo tante volte quanto indicato dal primo:

```
public class Calculator {

    public int multiply(final int a, final int b) {
        int res = 0;
        for (int i = 0; i < b; i++) {
            res += a;
        }
        return res;
    }
}
```

Supponiamo di utilizzare questa versione, per poi accorgerci che in alcuni casi non funziona. Pensiamo per esempio al caso in cui `a` e/o `b` fossero negativi. Decidiamo allora di modificare la classe nel seguente modo:

```
public class Calculator {

    public int multiply(final int a, final int b) {
        return a * b;
    }
}
```

In questo caso chi prima utilizzava il metodo `multiply()` sull'oggetto `Calculator`, potrà continuare a farlo anche dopo, senza accorgersi della modifica. Quello descritto è in realtà qualcosa di più del concetto di incapsulamento, in quanto ci permette di definire una struttura che in Java è di fondamentale importanza, ovvero l'interfaccia. Se osserviamo la classe precedente, notiamo come ciò che interessa al potenziale utilizzatore di un oggetto è *cosa fa* e non *come lo fa*. Cosa un oggetto è in grado di fare può essere descritto attraverso un'interfaccia definita nel seguente modo:

```
public interface ICalculator {

    int multiply(int a, int b);

}
```

Si tratta di un modo per elencare una serie di operazioni che un insieme di oggetti è in grado di assicurare. Quando una classe prevede tra le sue operazioni quelle descritte da un'interfaccia, si dice che "implementa" tale interfaccia. L'implementazione precedente può quindi essere definita nel seguente modo:

```
public class MyCalculator implements ICalculator {

    @Override
    public int multiply(int a, int b) {
        return a * b;
    }
}
```

Ma quale vantaggio si ha nel definire un'interfaccia e quindi un insieme di possibili implementazioni? Si tratta del concetto forse più importante della programmazione a oggetti: il *polimorfismo*. Ogni oggetto interessato a utilizzare un `ICalculator` (abbiamo utilizzato l'iniziale "I" per indicare che si tratta di un'interfaccia) avrebbe potuto scrivere al proprio interno del codice di questo tipo:

```
MyCalculator calc = new MyCalculator();
int res = calc.multiply(10,20);
```

Il vantaggio nell'utilizzo dell'interfaccia sarebbe stato nullo, in quanto quello che possiamo chiamare client (inteso come colui che utilizza), sa esattamente chi è l'oggetto a cui

andrà a chiedere il servizio. Abbiamo visto in precedenza che il nostro principale nemico è la dipendenza: possiamo minimizzarla facendo sì che i diversi oggetti conoscano il meno possibile l'uno dell'altro, in termini di implementazione. Al nostro client interessa solamente effettuare una moltiplicazione e quindi ha bisogno di un oggetto in grado di poterla effettuare. L'insieme di tutti questi oggetti può essere espresso attraverso un riferimento di tipo `ICalculator` e quindi il codice precedente può diventare:

```
ICalculator calc = new MyCalculator();
int res = calc.multiply(10,20);
```

In questo caso il client conosce però chi è il componente che eseguirà la moltiplicazione, per cui il miglioramento non è poi così grande. Si può fare di meglio con codice del tipo:

```
public class Client {

    private ICalculator mCalculator;

    public void setCalculator(final ICalculator calculator) {
        this.mCalculator = calculator;
    }

    public void doSomething() {
        mCalculator.multiply(10, 20);
    }

}
```

Qui il nostro client non sa chi è il particolare `ICalculator`, ma confida nel fatto che qualcuno dall'esterno gli passi un riferimento attraverso l'invocazione del metodo `setCalculator()`. L'aspetto importante si può riassumere nella possibilità di eseguire un'assegnazione del tipo:

```
ICalculator calculator = new MyCalculator();
```

Attraverso un riferimento di tipo `ICalculator` possiamo referenziare una qualsiasi implementazione dell'omonima interfaccia, qualunque essa sia. Il tipo del riferimento ci dice che cosa si può fare, mentre l'oggetto referenziato ci dice come questo viene implementato. Il fatto che il client utilizzi un riferimento di tipo `ICalculator` esprime questo disinteresse verso il "chi", ma l'interesse verso il "cosa".
Ma in Android dove viene utilizzato il polimorfismo? A dire il vero la piattaforma non utilizza questa importantissima caratteristica nel modo appena descritto, anche se il concetto di interfaccia e relativa implementazione rimane fondamentale. Vedremo, per esempio, cosa succede nella creazione dei *service*. Si tratta di qualcosa che sarà bene utilizzare nel codice delle nostre applicazioni come faremo in questo libro.
Il concetto forse più importante in Android è quello che, nello sviluppo enterprise, è visto come un nemico, ovvero l'*ereditarietà*. Quella che spesso si chiama *implementation inheritance* (a differenza di quello visto per le interfacce di *interface inheritance*) rappresenta infatti il vincolo di dipendenza ancora più forte. Come vedremo in questo libro, la rea-

lizzazione di un'applicazione Android prevede la creazione di alcuni componenti come specializzazioni di classi esistenti, come `Activity` e `Service`. Questo si rende necessario per permettere all'ambiente di gestirne il ciclo di vita attraverso opportuni metodi di *callback*. A mio parere questo rappresenta un problema che poteva essere affrontato in modo diverso attraverso la definizione di un *delegate*, come avviene in altri sistemi, primo fra tutti iOS. Una delle regole principali della programmazione a oggetti dice infatti "Composition over (implementation) inheritance", ovvero è meglio utilizzare piuttosto che estendere. È come se la nostra classe `Client` precedente avesse dovuto estendere la classe `MyCalculator` invece di utilizzarla attraverso un riferimento di tipo `ICalculator` come abbiamo fatto nell'esempio.
L'ereditarietà, in Java, presenta inoltre un problema relativo al fatto che è singola. Ogni classe Java può estendere al più un'unica classe (e lo fa sempre almeno con la classe `Object`) e implementare un numero teoricamente illimitato di interfacce. In Android esistono diverse librerie che hanno l'esigenza di "attaccarsi" al ciclo di vita dei componenti e in particolare delle `Activity`. Questo porta a dover estendere spesso le classi del framework. Andando sul concreto, supponiamo di voler utilizzare la *Compatibility Library*, una libreria messa a disposizione da Google inizialmente per poter usare i `Fragment` anche in versioni precedenti la 3.0, nella quale sono stati ufficialmente introdotti, per poi evolvere offrendo strumenti sempre più utili, come alcuni dei componenti grafici che utilizzeremo nella nostra applicazione e che si rifanno ai concetti di *Material Design*.

NOTA

I *Fragment* verranno affrontati più avanti nel testo e ci permetteranno una più granulare organizzazione dei layout, in modo da semplificarne l'utilizzo in display di dimensione diverse, come per esempio smartphone e tablet.

Per utilizzare i `Fragment`, le nostre `Activity` dovranno estendere una classe che si chiama `FragmentActivity`. Ma cosa succede nel caso in cui la nostra classe estendesse già un'altra classe? Questo sarebbe comunque un problema facilmente risolvibile, in quanto basterebbe risalire nella nostra gerarchia fino alla classe che estende `Activity` sostituendola con `FragmentActivity`. Il problema si ha però quando si vogliono utilizzare anche altri tipi di classi che offrono altri servizi trasversali, come potrebbe essere quello di *Dependency Injection*. In generale si tratta di strumenti che basano il loro funzionamento sulla gestione del ciclo di vita della `Activity`, eseguendo determinate operazioni in corrispondenza di alcuni metodi di callback cui si accede attraverso la *implementation inheritance*.

Il delegation model

Un concetto strettamente legato a quello di polimorfismo descritto nel paragrafo precedente è quello di *delegation model*, che sta alla base della gestione degli eventi in Java e quindi anche in Android, con qualche modifica per le convenzioni. Innanzitutto cerchiamo di descrivere il problema, che è quello di un evento generato da una sorgente e da un insieme di oggetti che ne sono interessati e che ne ricevono in qualche modo la notifica. Serve un meccanismo tale per cui la sorgente dell'evento non sappia a priori chi sono i `Listener`, i quali potrebbero essere istanze di classi qualunque.

NOTA

È bene sottolineare come il concetto di *Listener* non sia nuovo, ma si rifaccia semplicemente a un'implementazione di un famoso *GoF Pattern* che si chiama *Observer* (`https://en.wikipedia.org/wiki/Observer_pattern`).

Da qui capiamo come il concetto alla base di tutto sia quello di *interfaccia*. Facciamo un esempio concreto che prende come riferimento un evento che chiamiamo `Tic`. Se seguissimo le convenzioni descritte dalle specifiche JavaBeans, che di fatto hanno dato origine a questo meccanismo in Java, il primo passo consisterebbe nella creazione di una classe di nome `TicEvent` in grado di incapsulare tutte le informazioni dell'evento.
Nel caso standard, questa classe dovrebbe estendere la classe `java.util.EventObject`, che descrive la caratteristica che tutti gli eventi devono per forza avere, ovvero una sorgente che li ha generati. In Android questo non avviene e la sorgente dell'evento viene spesso passata come secondo parametro nei metodi di notifica. In ogni caso la sorgente è responsabile della memorizzazione dei `Listener` e quindi della notifica dell'evento. I `Listener`, come abbiamo detto, dovranno essere visti dalla sorgente come se fossero tutti dello stesso tipo, da cui la necessità di creare un'interfaccia che in Android sarà del tipo:

```
public interface OnTicListener {

    void onTick(View src, TicEvent event);

}
```

Tutti gli oggetti interessati all'evento dovranno quindi implementare questa interfaccia e il proprio metodo `onTick()`, all'interno del quale forniranno la propria elaborazione delle informazioni ricevute. Serve ora un meccanismo per informare la sorgente dell'interesse verso l'evento. Le specifiche JavaBeans prevedono di definire metodi del tipo:

```
public void addTickListener(TicListener listener);

public void removeTickListener(TicListener listener);
```

Nel caso Android, ma *mobile* in genere, diventano semplicemente:

```
public setOnTickListener(OnTickListener listener);
```

A parte la diversa convenzione del prefisso `On` sul nome dell'interfaccia, notiamo come il prefisso `add` del metodo sia stato sostituito dal prefisso `set`. Questo sta a indicare che le sorgenti degli eventi sono in grado di gestire un unico listener.

NOTA

In ambito desktop la possibilità di gestire un numero qualunque di *Listener* ha portato a diversi problemi di performance oltre che provocare spesso dei *deadlock*.

Proprio per il fatto che al più esiste un unico `Listener`, il metodo `remove` è stato eliminato; sarà quindi sufficiente impostare il valore `null` attraverso il precedente metodo `set`.

Durante lo sviluppo della nostra applicazione avremo modo di vedere quali eventi gestire e come. Per farlo dobbiamo comunque studiare un altro importante concetto, che è quello delle classi interne, come vedremo nel prossimo paragrafo.

Le classi interne

Le classi interne rappresentano una funzionalità molto interessante che è stata introdotta solamente a partire dalla versione 1.1 del JDK. Nella scrittura del codice ci si era infatti resi conto della mancanza di uno strumento che permettesse di creare determinate classi con visibilità limitata a quella di una classe esterna, che condividesse con la prima alcuni dati o che semplicemente fosse legata ad essa da un insieme di considerazioni logiche. Ci si è accorti che sarebbe stato molto utile poter definire una classe all'interno di un'altra, in modo da condividerne tutte le proprietà senza dover creare classi con costruttori o altri metodi con moltissimi parametri. Più di una funzionalità del linguaggio, quella delle classi interne è quindi relativa al compilatore. A tale proposito possiamo pensare a quattro diversi tipi di classi interne ovvero:

- classi e interfacce top level;
- classi membro;
- classi locali;
- classi anonime.

Vediamo di descriverle brevemente con qualche esempio.

Classi e interfacce top level

Questa categoria di classi interne descrive dei tipi che non hanno caratteristiche differenti dalle classi che le contengono, se non per il fatto di essere legate a esse da una relazione logica di convenienza, come può essere, per esempio, quella di appartenere a uno stesso concetto (contenitore e contenuto) o di riguardare uno stesso aspetto della programmazione (I/O, networking e così via). Queste classi interne sono anche dette statiche, in quanto ci si riferisce a esse utilizzando il nome della classe che le contiene allo stesso modo di una proprietà statica e della relativa classe. A tale proposito ci ricolleghiamo a quanto detto in relazione alla gestione degli eventi e definiamo la seguente classe:

```
public class EventSource {

    /**
    * Static inner class for the information about an event
    */
    public static class MyEvent {

        private final EventSource mSrc;

        private final long mWhen;

        private MyEvent(final EventSource src, final long when) {
            this.mSrc = src;
```

```
            this.mWhen = when;
        }

        public EventSource getSrc() {
            return mSrc;
        }

        public long getWhen() {
            return mWhen;
        }

    }

    /**
    * Inner class for the Listener of the event
    */
    public interface EventListener {

        void eventTriggered(MyEvent event);

    }

    private EventListener mListener;

    public void setEventListener(final EventListener listener) {
        this.mListener = listener;
    }

    public void notifyEvent() {
        if (mListener != null) {
            final MyEvent myEvent = new MyEvent(this,
                    System.currentTimeMillis());
            mListener.eventTriggered(myEvent);
        }
    }

}
```

La nostra classe si chiama `EventSource` e rappresenta la sorgente di un evento che abbiamo descritto attraverso una classe interna statica `MyEvent`. Notiamo innanzitutto come si tratti di una classe statica, che quindi non può accedere a membri non statici della classe esterna. Se provassimo a utilizzare, per esempio, la variabile `mListener`, otterremmo un errore di compilazione. Da notare come la classe `MyEvent` disponga di un costruttore privato, che è comunque accessibile dalla classe esterna, in quanto all'interno dello stesso file, come abbiamo detto all'inizio di questo capitolo in relazione ai modificatori di visibilità. Questo ci permette di limitare la creazione di istanze di `MyEvent` all'unica sua sorgente, come è corretto anche dal punto di vista logico.
Attraverso la definizione dell'interfaccia interna `EventListener` abbiamo quindi definito l'interfaccia che l'eventuale `Listener` dell'evento dovrà implementare. Infine abbiamo creato il metodo per la registrazione del `Listener` e un metodo privato che la sorgente utilizzerà per notificare l'evento all'eventuale `Listener`. In questa fase è importante notare

come le classi in gioco siano, relativamente al package in cui abbiamo definito la classe esterna, le seguenti:

```
EventSource
EventSource.MyEvent
EventSource.EventListener
```

Il nome completo di questo tipo di classi interne comprende anche il nome della classe esterna. La creazione di un'istanza di una di queste classi dall'esterno dovrà contenere tale nome del tipo, sempre nel caso in cui essa fosse comunque visibile (cosa che nel nostro esempio precedente non avviene):

```
EventSource.MyEvent myEvent = new EventSource.MyEvent();
```

Quindi la regola generale è

```
ExtClass.IntClass a = new ExtClass.IntClass();
```

Questo significa anche che un eventuale `Listener` dovrà essere del tipo:

```
public class MyListener implements EventSource.EventListener {

    @Override
    public void eventTriggered(MyEvent event) {
    }

}
```

e non semplicemente

```
public class MyListener implements EventListener {

    @Override
    public void eventTriggered(MyEvent event) {
    }

}
```

Questo a meno di non essere nello stesso *package* o di utilizzare una delle nuove feature di Java 5 che prevede la possibilità di importare i membri statici delle classi attraverso un nuovo tipo di `import`, che comunque consiglio di utilizzare con cura a favore di una migliore leggibilità del codice, in quanto si intenderebbe una visibilità di `default` o `package` che rappresenterebbe in questo caso una diminuzione di visibilità, cosa che non è possibile.

```
public interface EventListener {

    void eventTriggered(MyEvent event);

}
```

Le implementazioni sono del tipo:

```
public class MyListener implements EventListener {

    @Override
    public void eventTriggered(MyEvent event) {
    }

}
```

ovvero vi è il modificatore di visibilità `public`. In realtà nell'interfaccia questo modificatore è implicito. Non avrebbe infatti senso definire un'operazione di un'interfaccia come non pubblica. La classe quindi non potrebbe essere definita come:

```
public class MyListener implements EventListener {

    @Override
    void eventTriggered(MyEvent event) {
    }

}
```

in quanto si intenderebbe una visibilità di `default` o `package` che rappresenterebbe in questo caso una diminuzione di visibilità, cosa che non è possibile.

Classi membro

Le classi viste nel paragrafo precedente permettono di creare un legame logico del tipo contenitore/contenuto o sorgente-evento/evento e così via. La seconda tipologia di classi interne che ci accingiamo a studiare è quella delle classi membro, che sono di fondamentale importanza, specialmente in ambiente Android, dove vengono utilizzate moltissimo. In generale una *classe membro* è una classe che viene definita sempre all'interno di una classe esterna. Questa volta però le istanze della classe interna sono legate a particolari istanze della classe esterna. Anche qui ci aiutiamo con un esempio:

```
public class UserData {

    private final String mName;

    public UserData(final String name) {
        this.mName = name;
    }

    public class Printer {

        public void printName() {
            System.out.println("User is " + mName);
        }
    }
}
```

La classe esterna si chiama `UserData` e prevede la definizione di un proprio attributo che ne descrive il nome. Abbiamo poi definito una classe membro interna, che abbiamo chiamato `Printer`, che stampa il valore dell'attributo `mName` della classe esterna.
Come il lettore potrà verificare, il codice viene compilato con facilità. La domanda che ci poniamo riguarda il modo in cui associare un'istanza di `Printer` a una precisa istanza di `UserData`. La soluzione è nel metodo `main` che abbiamo aggiunto:

```
public static void main(String[] args) {
    final UserData pippo = new UserData("pippo");
    final Printer pippoPrinter = pippo.new Printer();
    final UserData pluto = new UserData("pluto");
    final Printer plutoPrinter = pluto.new Printer();

    // We print data
    pippoPrinter.printName();
    plutoPrinter.printName();
}
```

Notiamo come siano state create due istanze della classe `UserData` e quindi, a partire da queste, le relative istanze della classe interna `UserDara.Printer` utilizzando la seguente sintassi:

```
<istanza classe esterna>.new ClasseInterna();
```

Eseguendo il main è facile notare come ciascuna istanza della classe `Printer` porti con sé il valore della proprietà `mName` dell'istanza a cui è stata associata. Come vedremo durante lo sviluppo della nostra applicazione, l'utilizzo di questo tipo di classi può avere ripercussioni importanti nella gestione della memoria.
Come abbiamo detto in precedenza, le classi interne sono una feature del compilatore, il quale si preoccupa di aggiungere all'istanza della classe interna un riferimento all'istanza della classe esterna che lo potrebbe tenere in vita e quindi impedire al *garbage collector* di liberarne la memoria. È un primo esempio di come sia importante tenere in considerazione aspetti legati alla memoria anche nel caso di operazioni all'apparenza innocue, come la creazione di un'istanza di una classe interna.
A dire il vero, in Android non utilizzeremo quasi mai questa sintassi, in quanto creeremo le istanze delle classi interne direttamente nella classe esterna, come vedremo nel prossimo paragrafo.

Classi locali

Proseguiamo la descrizione delle classi interne con le *classi locali*, che, analogamente a quanto avviene per le variabili, si dicono tali perché definite all'interno di un blocco. Sono classi che si possono definire e utilizzare all'interno di un metodo e quindi possono accedere alle relative variabili o parametri. Anche in questo caso facciamo un esempio, riprendendo le classi statiche precedenti:

```
public class LocalClassTest {
```

```
    private EventSource mEventSource1;
    private EventSource mEventSource2;

    public LocalClassTest() {
        mEventSource1 = new EventSource();
        mEventSource2 = new EventSource();
    }

    public void useStaticClass() {
        final int num = 0;
        class Local implements EventListener {

            @Override
            public void eventTriggered(MyEvent event) {
                System.out.println("Event " + num);
            }
        }
        mEventSource1.setEventListener(new Local());
        mEventSource2.setEventListener(new Local());
    }
}
```

Abbiamo creato una classe che definisce due attributi di tipo `EventSource`, che ricordiamo essere sorgenti di un evento. La parte interessante è invece all'interno del metodo `useStaticClass()`, nel quale definiamo una classe locale che abbiamo chiamato `Local` che poi istanziamo per registrare due `Listener`. È di fondamentale importanza notare come la classe `Local` utilizzi al suo interno il valore di una variabile locale che abbiamo chiamato `num`, la quale deve essere necessariamente definita come `final`, il che la rende di fatto costante. Questo è un requisito fondamentale nel caso delle classi locali. Per capire il perché consideriamo il seguente metodo:

```
public EventListener externalise() {
    final int num = 0;
    class Local implements EventListener {

        @Override
        public void eventTriggered(MyEvent event) {
            System.out.println("Event " + num);
        }
    }
    return new Local();
}
```

Esso restituisce un oggetto di tipo `EventSource.EventListener`, che nel nostro caso è l'istanza della classe locale `Local` creata all'interno del metodo e che utilizza il riferimento a `num`. Essendo una variabile locale, ci si aspetta che essa viva quanto l'esecuzione del metodo. Nel nostro caso, se non ne fosse creata una copia, avrebbe una vita molto più lunga, legata di fatto alla vita dell'istanza restituita. Lo stesso non vale nel caso in cui `num` fosse una variabile d'istanza.

Classi anonime

Eccoci finalmente al tipo di classi interne più utili, ovvero le *classi anonime*. Per spiegare di cosa si tratta torniamo alla nostra `EventSource` e supponiamo di voler registrare un `Listener`. In base a quello che conosciamo adesso, dovremmo scrivere questo codice:

```
EventSource source = new EventSource();
class MyListener implements EventListener {

    @Override
    public void eventTriggered(MyEvent event) {
        // Do something
    }
}
source.setEventListener(new MyListener());
```

A questo punto ci domandiamo se valga la pena di definire una classe, e quindi una sua istanza, quando quello che ci interessa è l'esecuzione del metodo `eventTriggered()` a seguito di un evento. È in situazioni come questa che ci vengono in aiuto le classi anonime, che ci permettono di scrivere il seguente codice equivalente:

```
EventSource source = new EventSource();
source.setEventListener(new EventListener(){

    @Override
    public void eventTriggered(MyEvent event) {
        // Do something
    }
});
```

Notiamo come si possa creare "al volo" un'istanza di una classe che implementa una particolare interfaccia, semplicemente utilizzando la sintassi

```
new Interfaccia() {
    // Implementazione operazioni interfaccia
};
```

Si parla di classe anonima in quanto la classe, in realtà, non viene definita e quindi non ha un nome. Una sintassi analoga si può utilizzare con le classi, ovvero

```
new MyClass() {
    // Override dei metodi della classe
};
```

Questa volta il significato è quello di creare un'istanza di un'ipotetica classe che estende `MyClass` e di cui si può fare l'override di alcuni metodi. Durante la realizzazione del nostro progetto vedremo spesso questo tipo di classi, non solo nella gestione degli eventi.

Generics

L'argomento che ci accingiamo a trattare ora è quello dei `Generics`, che sono stati introdotti dalla versione 5 della piattaforma Java e ci permettono di creare delle classi parametrizzate rispetto ai tipi che esse gestiscono. Anche qui ci aiutiamo con qualche esempio che utilizza delle `collection` e che rappresentano alcune delle strutture dati più importanti in Java. Una di queste è per esempio la `List`, descritta dall'omonima interfaccia del `package java.util`. Essa descrive, in sintesi, una struttura dati che contiene oggetti sequenziali e può essere implementata in modi diversi. Pensiamo, per esempio, a un'implementazione che utilizza un array descritto dalla classe `ArrayList` o che prevede l'utilizzo di una lista concatenata descritta dalla classe `LinkedList`. In ogni caso, in una lista possiamo aggiungere elementi e poi accedere a tali elementi attraverso il concetto di indice. Supponiamo di scrivere le seguenti righe di codice:

```
List list = new LinkedList();
list.add(1);
list.add(2);
list.add(3);
```

Queste permettono di creare una lista implementata come lista concatenata, alla quale aggiungiamo tre oggetti. Quest'ultima affermazione potrebbe trarre in inganno se non si conoscesse un'altra importante caratteristica di Java 5 che prende il nome di *autoboxing*. Essa prevede che gli oggetti di tipo primitivo vengano automaticamente convertiti in istanze dei corrispondenti tipi *wrapper*. Nel nostro caso gli oggetti inseriti nella lista sono in effetti oggetti di tipo `Integer`.
La nostra lista può però contenere istanze di `Object`, per cui se aggiungessimo questa istruzione non ci sarebbe alcun errore di compilazione né di esecuzione:

```
list.add("four");
```

Supponiamo ora di voler estrarre i precedenti valori con il seguente codice:

```
for (int i = 0; i < list.size(); i++) {
    final Integer item = (Integer) list.get(i);
    System.out.println("Value " + item);
}
```

Sapendo che gli elementi nella lista sono di tipo `Integer` non facciamo altro che estrarli, farne il cast e quindi "stamparli" a video.

> **NOTA**
>
> In realtà il cast sarebbe inutile, in quanto nella stampa concateniamo il valore estratto dalla lista con la stringa che si ottiene dall'invocazione del metodo *toString()*. Grazie al polimorfismo il risultato sarebbe esattamente lo stesso.

Se però andiamo a eseguire il nostro ciclo `for`, che comunque viene compilato con successo, otterremmo la seguente eccezione, dovuta al fatto che il quarto elemento inserito non è un `Integer` ma una `String`:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to
java.lang.Integer
    at uk.co.maxcarli.introjava.generics.Main.main(Main.java:19)
```

Abbiamo ottenuto, in fase di esecuzione dell'applicazione, un errore che il compilatore non ci aveva messo in evidenza. È in questo contesto che i `generics` ci vengono in aiuto dandoci la possibilità di creare non una semplice `List`, ma un qualcosa di più, ovvero una `List` di `Integer` che possiamo esprimere nel seguente modo:

```
final List<Integer> listi = new LinkedList<Integer>();
listi.add(1);
listi.add(2);
listi.add(3);
```

In questo caso un'istruzione del seguente tipo porterebbe a un errore in compilazione, in quanto non si può inserire una `String` in una lista di `Integer`:

```
list.add("four");
```

I vantaggi non si hanno solamente nell'inserimento, ma soprattutto nell'estrazione dei valori. Da una `List` di `Integer` possiamo infatti estrarre solamente `Integer`, per cui non ci sarà bisogno di alcuna operazione di *cast*; di conseguenza, un'istruzione come la seguente è perfettamente legale:

```
Integer a = list.get(2);
```

I `generics` sono comunque molto di più. Supponiamo di avere due liste: una di `Integer` e una di `String`, come nel seguente codice:

```
List<Integer> list = new LinkedList<Integer>(); list.add(1); list.add(2); list.add(3);
List<String> list2 = new LinkedList<>(String);
list2.add("one");
list2.add("two");
list2.add("three");
```

Pensiamo a un metodo che ne permetta la visualizzazione. Il primo tentativo di un programmatore inesperto potrebbe essere questo:

```
public static void print(List<Integer> listi) {
    for (Integer item : listi) {
        System.out.println("Value " + item);
    }
}

public static void print(List<String> lists) {
    for (String item : lists) {
        System.out.println("Value " + item);
    }
}
```

Vengono realizzati due metodi `print()` che si differenziamo per il tipo di parametro, ciascuno associato a una lista di elementi diversa. Questo primo tentativo fallisce inesorabilmente, per il semplice fatto che per il compilatore i due metodi sono esattamente uguali. Come altre funzionalità, anche quella dei `generics` è legata al compilatore, che crea in entrambi i casi una `List` di oggetti. La soluzione più comune sarebbe cambiare il nome dei metodi e utilizzarli nel seguente modo:

```
public static void printIntegers(List<Integer> listi) {
    for (Integer item : listi) {
        System.out.println("Value " + item);
    }
}

public static void printStrings(List<String> lists) {
    for (String item : lists) {
        System.out.println("Value " + item);
    }
}
```

In questo caso il codice verrebbe compilato ed eseguito in modo corretto, ma sicuramente non farebbe fare una bella figura a Java. Per ogni tipo diverso di `List` si dovrebbe creare un metodo di nome differente che fa esattamente la stessa cosa degli altri.

NOTA

A tale proposito approfittiamo per far notare l'utilizzo di *enhanced for*, che permette di scorrere gli elementi di una collection o di un array senza l'utilizzo di un indice.

In realtà la soluzione esiste ed è rappresentata dal seguente metodo:

```
public static void print(List<?> list) {
    for (Object item : list) {
        System.out.println("Value " + item);
    }
}
```

Qui si utilizza una notazione all'apparenza strana, che ci permette di definire una `List` di `unknown`. Attraverso un riferimento di tipo `List<?>` possiamo così referenziare una lista di un tipo qualsiasi, ma a un prezzo: da essa possiamo estrarre oggetti, ma non inserirne. Vediamo di capirne il motivo, prendendo un esempio molto classico che utilizza le classi `Animal` e `Dog`, dove la seconda è una specializzazione della prima. In sintesi, un cane è un animale e quindi `Dog is a Animal`. Questa affermazione ci permette anche di scrivere questa istruzione:

```
Animal a = new Dog();
```

Attraverso un riferimento di tipo `Animal` possiamo referenziare un `Dog`. Attraverso questo riferimento potremo vedere solamente quelle caratteristiche di un `Dog` che lo identificano come animale. Per essere precisi potremo scrivere:

```
a.eat();
```

Questo perché ogni animale, e quindi anche il cane, è in grado di mangiare (eat) ma non potremo scrivere:

```
a.bark()
```

Sebbene un cane sia in grado di abbaiare (bark) e l'oggetto referenziato dalla variabile a sia effettivamente un cane, stiamo osservando tale oggetto attraverso un riferimento di tipo Animal e non tutti gli animali abbaiano. Ora ci chiediamo se invece un'istruzione di questo tipo può essere valida, ovvero se un array di Dog sia un array di Animal:

```
Animal[] a = new Dog[10];
```

Qui iniziamo ad avere qualche difficoltà: l'istruzione precedente compila perfettamente ma presenta un grosso problema che cerchiamo di spiegare. In questo caso a è un riferimento a un array di Animal che in questo momento sta referenziando un array di Dog. Questo significa che un'istruzione del tipo seguente è perfettamente valida e infatti viene compilata ed eseguita correttamente.

```
a[0] = new Dog();
```

Consideriamo ora però questa istruzione:

```
a[1] = new Cat();
```

Anche Cat è una classe che estende Animal, in quanto anche un gatto è un animale. Anche qui il codice viene compilato, in quanto a[1] è un riferimento a un oggetto di tipo Animal che nell'istruzione precedente riceve un valore con un riferimento a un Cat che è un Animal. Il problema sta però nell'oggetto referenziato, che è un array di Dog, per cui stiamo cercando di inserire un Cat in un array di Dog, provocando un errore in esecuzione. In particolare l'errore è il seguente:

```
Exception in thread "main" java.lang.ArrayStoreException: uk.co.maxcarli.introjava.generics.
Cat at uk.co.maxcarli.introjava.generics.Main.main(Main.java:30)
```

Come nel caso delle liste precedenti, si ha un errore in fase di esecuzione che non si era previsto in fase di compilazione. Facciamo allora un ulteriore passo avanti e chiediamoci se la prossima assegnazione è corretta, se viene compilata e, in caso affermativo, se viene eseguita correttamente:

```
List<Animal> a = new LinkedList<Dog>();
```

Ci chiediamo se una List di Animal sia una List di Dog. Questa volta la risposta è *no*. Una lista di cani non è una lista di animali. Questo perché, se lo fosse, analogamente a quanto visto prima, attraverso il riferimento alla lista di animali riusciremmo a inserire un gatto all'interno di una lista di cani. Questo fatto viene però riconosciuto dal compilatore, che ci notifica un errore che possiamo correggere, ma come? Se abbiamo bisogno di un tipo

di riferimento che ci permetta di referenziare sia una lista di cani sia una lista di gatti o di altro, la sintassi da utilizzare è questa:

```
List<?> a = new LinkedList<Dog>();
```

Quindi, le seguenti istruzioni diventano perfettamente lecite:

```
List<?> listaAnimali = new LinkedList<Dog>();
listaAnimali = new LinkedList<Animal>();
listaAnimali = new LinkedList<Cat>();
```

C'è un prezzo: analogamente a quanto visto nel caso precedente, non possiamo inserire valori ma solamente estrarne. Questo perché, per quanto detto, se potessimo eseguire inserimenti, avremmo la possibilità di aggiungere, attraverso l'astrazione, un oggetto di un tipo all'interno di una lista di un altro tipo. Tramite una `List<?>` non potremmo inserire un `Dog`, in quanto la lista referenziata poteva essere di `Cat` o di un qualunque altro tipo. Bene, ma se possiamo estrarre oggetti, di che tipo sarà l'oggetto estratto? Qui la risposta è un salomonico "non lo sappiamo", ma non lo ignoriamo completamente. Qualunque sia il tipo dell'oggetto che andiamo a estrarre si tratterà sicuramente di qualcosa che possiamo associare a un riferimento di tipo `Object`, in quanto tutti gli oggetti in Java sono istanze di una classe che, direttamente o indirettamente, estende la classe `Object`. Ecco che un'istruzione di questo tipo è perfettamente valida:

```
List<?> lista = ...;
Object item = lista.get(0);
```

Possiamo però fare ancora meglio, perché nel nostro caso non stiamo lavorando con classi qualunque, ma con animali, ovvero oggetti che estendono la classe `Animal`. Possiamo quindi dire che, degli oggetti che andiamo a estrarre dalla lista, qualcosa, alla fine, conosciamo. Non sappiamo esattamente di che tipo sono, ma sappiamo sicuramente che sono animali. Il precedente metodo `print()` può essere scritto nel seguente modo:

```
public static void print(List<? extends Animal> list) {
    for (Animal item : list) {
        System.out.println("Value " + item);
    }
}
```

Attraverso la notazione `List<? extends Animal>` indichiamo un riferimento a una lista di oggetti che non sappiamo ancora di che tipo siano (se sono `Dog`, `Cat` o altro) ma che sappiamo di certo sono specializzazioni della classe `Animal`. Questo ci permette da un lato di restringere i tipi di parametri che possono essere passati al metodo e dall'altro di avere qualche informazione in più sugli oggetti che possiamo estrarre, che ora, qualunque cosa siano, possono sicuramente essere assegnati a un riferimento di tipo `Animal` come abbiamo fatto nel nostro `enhanced for`. In questo caso possiamo anche inserire qualcosa? Purtroppo, la risposta è ancora *no*, perché sappiamo, sì, che si tratta di animali, ma non sappiamo di che tipo. Esiste ancora, per esempio, il pericolo di inserire `Dog` dove ci sono `Cat`. Ci chiediamo allora se esista una notazione simile alla precedente, ma che ci permetta di inserire oggetti in una lista. Certo: tale espressione è la seguente:

```
List<? super Animal> lista;
```

In questo caso il significato è quello di riferimento a una lista di `Animal` o di sue superclassi, che in questo caso può essere solo la classe `Object`. Per fare un esempio diverso, il riferimento seguente permette di referenziare una lista di `Dog` di `Animal` o di `Object`.

```
List<? super Dog> lista;
```

Prendendo sempre quest'ultimo esempio, in una lista come questa possiamo anche eseguire inserimenti, e precisamente possiamo inserire istanze di `Dog` o di sue classi figlie. Esse saranno in effetti sempre `Dog`, oppure `Animal` oppure `Object`, nel peggiore dei casi. Ma dove si utilizzano espressioni di questo tipo? Non entreremo nei dettagli, ma pensiamo per esempio al caso in cui dovessimo ordinare un insieme di questi oggetti attraverso un `Comparator`. Un particolare `Comparator<T>` è un oggetto in grado di riconoscere se un oggetto di tipo `T` è maggiore, uguale o minore di un altro dello stesso tipo. Supponiamo di voler creare, attraverso questo tipo di `Comparator`, un metodo di ordinamento di oggetti di tipo `T` contenuti in una lista. La firma di questo metodo, che si dice *generico*, sarà del tipo:

```
public <T> void sort(List<T> list, Comparator<T> comp)
```

La definizione di `<T>` all'inizio del metodo indica, appunto, che si tratta di un metodo con parametri generici e ci consente di descrivere la relazione che li lega. Il significato in questo caso è quello di un metodo che permette di ordinare oggetti di tipo `T` contenuti in una lista attraverso un `Comparator` in grado di ordinare oggetti dello stesso tipo `T`. Per stare sul concreto, una `List<Dog>` potrà essere ordinata solo se si ha a disposizione un `Comparator<Dog>`. In realtà sappiamo che un `Dog` è un `Animal`, per cui potremmo ordinare la lista di `Dog` anche se avessimo a disposizione un `Comparator<Animal>` o un `Comparator<Object>`. Un `Comparator<Animal>` sarebbe in grado di ordinare i `Dog` in quanto `Animal` perderebbe le sue caratteristiche di cane, ma questo va stabilito in fase di analisi del metodo che stiamo scrivendo. Nel caso in cui decidessimo di permettere questa cosa, la firma del metodo diventerebbe la seguente:

```
public <T> void sort(List<T> list, Comparator<? super T> comp)
```

Se poi volessimo ampliare ulteriormente, potremmo scrivere lo stesso metodo nel seguente modo:

```
public <T> void sort(List<? extends T> list, Comparator<? super T> comp)
```

Abbiamo quindi capito come i `generics` siano uno strumento molto potente e da usare a fondo nello sviluppo delle nostre applicazioni, più come utilizzatori che come realizzatori di classi generiche. Per completezza vogliamo concludere con un paio di esempi molto semplici e allo stesso tempo utili. Il primo riguarda la possibilità di creare una classe che funge da `Holder` di un oggetto di tipo `T`:

```
public class Holder<T> {

    private T mValue;
```

```
    public Holder(T value) {
        this.mValue = value;
    }

    public T getValue() {
        return mValue;
    }
}
```

Attraverso questo semplice codice abbiamo creato una classe immutabile di nome `Holder`, in grado di memorizzare il riferimento a qualunque oggetto di tipo `T`. Mediante questa classe possiamo scrivere righe di codice di questo tipo:

```
final Holder<Integer> iHolder = new Holder<Integer>(10);
Integer iValue = iHolder.getValue();
final Holder<String> sHolder = new Holder<String>("Hello");
String sValue = sHolder.getValue();
```

Notiamo come sia semplice e utile creare istanze di `Holder` che incapsulano valori di tipo diverso che poi è possibile ottenere attraverso il corrispondente metodo `getValue()`. L'ultima osservazione riguarda la `type inference`, che consiste nel far ritornare a un metodo un oggetto di tipo dipendente da quello della variabile a cui lo stesso viene assegnato. Andiamo anche qui sul concreto. Quando tratteremo le `Activity` vedremo come ottenere il riferimento a un elemento del corrispondente layout attraverso un metodo del tipo

```
protected View findViewById(int viewId);
```

Si tratta di un metodo che, dato un identificatore di tipo intero, restituisce una particolare specializzazione della classe `View`, che può essere un `Button`, un `TextView`, una `EditText` e così via. Vedremo come vi siano spesso delle istruzioni del tipo:

```
TextView output = (TextView) findViewById(R.id.output);
```

Quello che vogliamo fare è eliminare il fastidioso `cast` e fare in modo che il tipo di `View` restituita dal metodo `getView()` sia quello della variabile cui viene assegnato. Osserviamo la seguente implementazione di un metodo statico, che supponiamo essere della classe UI:

```
public <T extends View> findViewById(Activity activity, int id){
    final View container = activity.getWindow().getDecorView();
    return (T) container.findViewById(id);
}
```

Attraverso un metodo generico possiamo fare in modo di scrivere l'istruzione precedente nel seguente modo:

```
TextView output = UI.findViewById(activity, R.id.output);
```

Attenzione: si tratta solamente di una semplificazione che non garantisce che l'oggetto restituito sia effettivamente del tipo voluto. Lo stesso problema si aveva comunque anche nel caso precedente, ma così si evita il cast.

Annotation

L'idea che sta alla base delle *annotation* non è nuova, e risale alla prima versione di Java, del 1995. Ricordiamo, infatti, che una delle caratteristiche di questa piattaforma era quella di poter generare in modo automatico la documentazione delle classi, attraverso un tool che si chiamava *JavaDoc* e che prevedeva l'utilizzo di alcune notazioni del tipo `@params` o simili all'interno dei commenti del codice stesso. L'idea, iniziata con alcuni progetti open source, è stata quella di utilizzare questa stessa notazione come una specie di direttiva, che poteva poi essere utilizzata in diversi momenti del processo di sviluppo, ovvero scrittura del codice, build o esecuzione dell'applicazione. Sono nate quindi le *annotation*, integrate in Java dalla versione 5. Ma perché abbiamo deciso di trattare questo argomento? Semplicemente perché molti dei tool di sviluppo e framework, si basano sull'utilizzo di questi strumenti e Android Studio *in primis*.
Come abbiamo detto, le *annotation* possono essere utilizzate in fase di scrittura di codice, di build oppure a runtime. Un tipico esempio di utilizzo delle annotation in fase di scrittura del codice riguarda l'annotation `@Override` che abbiamo già incontrato in precedenza e che ci permette di ricevere subito conferma del fatto che si stia effettivamente facendo l'override di un metodo di una classe che la nostra estende (*implementation inheritance*) o l'implementazione di un'operazione di un'interfaccia (*interface inheritance*). Supponiamo di avere una classe molto semplice, che abbiamo chiamato `User`, la quale prevede solamente la definizione della proprietà `name`.

```
public class User {

    private final String mName;

    public User(final String name) {
        this.mName = name;
    }
}
```

Creiamo quindi un semplice metodo `main()`, nel quale creiamo un contenitore di tipo `Set`, rappresentato dalla sua implementazione `HashSet`, nel quale aggiungiamo due istanze di `User` che utilizzano lo stesso nome.

> **NOTA**
>
> A proposito, prima di proseguire notiamo una notazione introdotta con Java7, *new HashSet<>()* , che prevede di non specificare il tipo generico nel caso in cui questo sia evidente e derivato attraverso *inference*.

Se eseguiamo il nostro semplice programma, notiamo come venga prodotto il valore 2. Questo perché le due istanze di `User` vengono considerate istanze diverse e questo potrebbe essere esattamente quello che si vuole e che tutti gli `User`, anche con lo stesso nome, vengano considerati diversi. Ora supponiamo invece che quello che definisce uno `User` sia appunto il suo `name` e che quindi due User con lo stesso `name` vengano considerati lo stesso all'interno di un `Set` che ricordiamo non ammettere duplicati.

```
public static void main(String[] args) {
```

```
    final String NAME = "Pippo";
    final User user1 = new User(NAME);
    final User user2 = new User(NAME);
    final Set<User> set = new HashSet<>();
    set.add(user1);
    set.add(user2);
    System.out.println("Dimension " + set.size());
}
```

Il lettore avrà capito che il tutto si basa sul concetto di duplicato, ovvero sul come un `Set` capisce se un'istanza di un oggetto è uguale a quella di un altro. Questo ci permette di introdurre un altro aspetto fondamentale di Java, ovvero quello di gestione dell'uguaglianza tra due oggetti. La nostra classe è un `HashSet`, il quale prevede che due oggetti vengano considerati uguali attraverso l'utilizzo di due metodi di cui ogni oggetto Java dispone, perché definiti all'interno della classe `Object`. Si tratta dei metodi:

```
public int hashCode();
public boolean equals(Object obj);
```

Tutto il meccanismo di gestione dell'uguaglianza si basa sul fatto che se due oggetti sono uguali, allora debbano necessariamente restituire lo stesso valore di hashcode. Se però due oggetti hanno lo stesso valore di hashcode non è detto che siano uguali; l'effettiva uguaglianza verrà decisa dal risultato del metodo `equals()`. Questo fatto ha come conseguenza che se due oggetti sono `equals()` allora i relativi metodi `hashCode()` devono restituire lo stesso valore. Il concetto è quindi lo stesso alla base della gestione di una struttura dati fondamentale, ovvero la `Hashtable` (`https://en.wikipedia.org/wiki/Hash_table`). Tornando al nostro esempio, che cosa dovremo fare quindi per fare in modo che due User con lo stesso nome vengano considerati uguali? La risposta è l'override dei due metodi citati, in modo da soddisfare il contratto appena descritto. Ci mettiamo al lavoro e implementiamo i precedenti metodi nel seguente modo, senza entrare nel dettaglio della regola utilizzata nell'implementazione, che il lettore può comunque vedere nel precedente link.

```
public int hashcode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((mName == null) ? 0 : mName.hashCode());
    return result;
}

public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    User other = (User) obj;
    if (mName == null) {
        if (other.mName != null)
            return false;
```

```
    } else if (!mName.equals(other.mName))
        return false;
    return true;
}
```

Eseguiamo quindi nuovamente l'applicazione ottenendo il seguente output:

```
Dimension 2
```

Ma come? Tutto questo lavoro per niente? Allora tutto quello descritto non è vero? In realtà nel precedente codice abbiamo commesso un errore, che è di difficile individuazione, ma che ha delle conseguenze molto gravi. Per scoprire quale, utilizziamo l'annotazione `@Override` nel seguente modo:

```
@Override
public int hashcode() {
    - - -
}

@Override
public boolean equals(Object obj) {
    - - -
}
```

Questo metterà in evidenza l'errore, ovvero che il metodo `hashcode()` non è il metodo `hashCode()` nel quale la C è maiuscola. Nel primo caso non avevamo eseguito l'override del metodo corretto, ma semplicemente definito un altro metodo di nome simile, ma per Java completamente diverso, che non veniva mai invocato. Non ci resta quindi che risolvere il problema nel seguente modo:

```
@Override
public int hashCode() {
    - - -
}

@Override
public boolean equals(Object obj) {
    - - -
}
```

Ora possiamo nuovamente eseguire il nostro programma ottenendo il seguente output:

```
Dimension 1
```

Quella che abbiamo visto è solamente una delle *annotation* che è possibile utilizzare in fase di scrittura del codice. Altre sono definite e gestite dal tool utilizzato nella scrittura del codice e permettono, appunto, di evitare situazioni spiacevoli e di difficile individuazione, come quella appena descritta. Durante la scrittura del codice ne vedremo altre che ci permetteranno, per esempio, di individuare riferimenti che sono `null` quando non dovrebbero.

Un aspetto che forse non abbiamo evidenziato abbastanza riguarda il fatto che ciascuna annotazione non è altro che quello che dice la parola stessa, ovvero un modo per annotare il codice con informazioni che devono poi essere lette ed elaborate da un altro componente, che si chiama *Annotation Processor*. Un'annotazione non è codice che viene eseguito, nel senso che non ha lo stesso significato di una qualunque altra istruzione. Nel caso dell'annotazione `@Override`, è il tool di sviluppo che ne interpreta il significato, verificando se effettivamente è applicato a un'operazione di cui ne viene fatto l'*override*. Nello specifico si tratta di un'annotazione che ha significato solo a livello di codice, che quindi potrebbe essere persa nel corrispondente *bytecode*. Altre annotazioni possono essere utilizzate da tool che permettono la generazione di codice, mentre altre ancora in fase di esecuzione. A dire il vero le annotazioni di quest'ultimo tipo non sono molto utilizzate in ambito Android, in quanto molto pesanti da un punto di vista delle performance. Per completezza concludiamo comunque questo capitolo realizzando una nostra annotazione che chiamiamo `@Singleton` e che ci permette di fare in modo che di una classe ne venga creata solamente un'istanza raggiungibile da ogni parte dell'applicazione (`https://en.wikipedia.org/wiki/Singleton_pattern`). Si tratta di un esercizio molto utile, che ci permetterà di vedere altri concetti relativi allo sviluppo in Java che, prima o poi, potrebbero essere utili nello sviluppo delle applicazioni Android. Il codice della nostra annotazione è il seguente:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(value = ElementType.TYPE)
public @interface Singleton {

    String name() default "singleton";

}
```

Come possiamo notare, essa si definisce in un modo che ricorda la definizione di un'interfaccia, con la differenza nell'utilizzo della parola chiave `@interface`. Se pensiamo al significato di un'interfaccia e a quello di un'annotazione possiamo infatti trovare un qualcosa di comune, ovvero entrambi rappresentano un modo per descrivere che una particolare classe soddisfa alcune proprietà. Nel caso dell'interfaccia si tratta di garantire la presenza di alcune operazioni, mentre nel caso dell'annotazione si tratta di indicare che, per esempio, si tratta di un *singleton*. Come possiamo vedere, anche le annotazioni possono essere annotate per indicarne alcune caratteristiche. La prima è proprio la *retain policy* cui abbiamo accennato in precedenza. Attraverso la definizione:

```
@Retention(RetentionPolicy.RUNTIME)
```

abbiamo specificato che la nostra annotazione verrà utilizzata a runtime. La seconda definizione:

```
@Target(value = ElementType.TYPE)
```

ci permette di indicare, invece, a quali elementi l'annotazione possa essere applicata. Nel nostro caso abbiamo voluto dire che la nostra annotazione potrà essere utilizzata solamente nella definizione di un tipo. Purtroppo non è possibile indicare solamente le

classi per cui la nostra annotazione potrà essere applicata anche a *enum*, anche se, in quel caso, si tratterebbe di una definizione superflua.

> **NOTA**
>
> In effetti una *enum* rappresenta un modo per indicare un numero specifico e finito di possibili istanze di una classe. In questo testo non tratteremo questo costrutto, per il semplice motivo che è molto poco performante in ambito Android e quindi ne è sconsigliato l'utilizzo.

Infine un'annotazione può disporre anche di alcune operazioni che, in realtà, permettono di specificare i suoi attributi. Ogni annotazione ha sempre un attributo di nome `value`, il quale può essere esplicitato o meno nel caso in cui fosse l'unico, come avvenuto nel nostro esempio per le annotazioni `@Retention` e `@Target`. Nel nostro caso abbiamo anche un attributo di nome `name`, che notiamo essere opzionale. L'obbligatorietà o meno di un attributo è indicata dalla presenza o meno della parola chiave `default` seguita, appunto, dal valore di default. Se non è presente, l'attributo è da considerarsi obbligatorio e deve quindi essere specificato in fase di utilizzo.
Ora proviamo a creare una nostra classe annotandola con `@Singleton` ottenendo quanto segue:

```
@Singleton(name = "my_class")
public class MyClass {

}
```

A questo punto il lettore potrebbe chiedersi: "E allora?". Beh, in effetti non abbiamo fatto assolutamente nulla, se non creato una *annotation* che abbiamo poi utilizzato nella definizione di una classe. Come abbiamo detto in precedenza serve un componente che utilizzi questa annotazione in relazione alle proprie funzionalità. Nel caso della nostra annotazione ci stiamo riferendo alla modalità con cui gli oggetti vengono creati. Per questo motivo abbiamo creato la classe `New` con il metodo statico `create()` alla quale deleghiamo la creazione delle istanze di una classe che passiamo come parametro.

```
public class New {

    private static Map<String, Object> OBJS = new HashMap<>();

    public static <T> T create(Class<T> clazz) throws Exception {
        final Singleton singleton = clazz.getAnnotation(Singleton.class);
        if (singleton == null) {
            // It's not a singleton
            return clazz.newInstance();
        } else {
            // We check if already created
            T cached = (T) OBJS.get(singleton.name());
            if (cached != null) {
                return cached;
            }
            cached = clazz.newInstance();
```

```
            OBJS.put(singleton.name(), cached);
            return cached;
        }
    }
}
```

Come possiamo notare nel codice evidenziato, attraverso il metodo `getAnnotation()` riusciamo a capire se la corrispondente classe è o meno annotata con `@Singleton`. Nel caso non lo fosse il valore restituito sarà `null`, per cui non faremo altro che crearne un'istanza usando il corrispondente metodo della classe `Class`.

> **NOTA**
>
> In Java ciascuna classe *T* è descritta da un'istanza della classe *Class<T>* che mette a disposizione una serie di metodi per eseguirne una *introspection*. Si tratta di istruzioni che in Android sono sconsigliate, in quanto degradano di molto le performance delle applicazioni.

Nel caso in cui la classe sia annotata, notiamo invece come sia stato invocato il metodo `name()` che restituirà il valore dell'attributo utilizzato per la classe annotata, che andremo quindi a utilizzare per salvare l'istanza in una `Map`. Lasciamo al lettore l'esecuzione del seguente metodo e la verifica del come un'istanza venga o meno riutilizzata a seconda che la classe utilizzata sia o meno annotata con `@Singleton`:

```
public static void main(String[] args) throws Exception {
    MyClass myclass1 = New.create(MyClass.class);
    MyClass myclass2 = New.create(MyClass.class);
    if (myclass1 == myclass2) {
        System.out.println("THE SAME");
    } else {
        System.out.println("NOT THE SAME");
    }
}
```

Conclusioni

In questo primo capitolo abbiamo posto le basi per poter affrontare il nostro progetto e iniziare a realizzare la nostra applicazione Android. Nella prima parte ci siamo occupati di che cos'è Android: la sua architettura, la sua relazione con Java e quindi i suoi componenti principali. Si è trattato di un'infarinatura di quello che ci attende nelle prossime pagine. Nella seconda parte del capitolo ci siamo dedicati al linguaggio Java e agli aspetti che ci potranno essere più utili in ambito Android. Ora siamo davvero pronti, per cui mettiamoci al lavoro.