Capitolo 1

Android e Java per Android

In questo capitolo

- **Cos'è Android**
- **Introduzione a Java**
- **Conclusioni**

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 per quasi scontato saranno i concetti di programmazione ad oggetti. Se il lettore si sente già a suo agio con il linguaggio potrà leggere solamente la prima parte e quindi andare direttamente al capitolo successivo. Per iniziare descriveremo la classica applicazione `Hello World`, che nasconde un numero elevatissimo di concetti che 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. Concluderemo quindi con la descrizione dei *generics*.

Di solito il primo capitolo di un libro di questo tipo descrive anche l'installazione dell'ambiente. Qui abbiamo deciso di rimandare al capitolo successivo per due ragioni principali. La prima riguarda il fatto che proprio in questi giorni è stato presentato alla Google IO un nuovo tool di sviluppo che si chiama Android Studio basato su IntelliJ. Il secondo motivo è invece legato all'elevata volatilità della procedura di installazione, che cambia di continuo. Per questa daremo il riferimento della documentazione ufficiale.

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, che si chiama Dalvik, 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. 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

Quando si introduce Android si utilizza sempre la famosa immagine mostrata nella Figura 1.1 che ne descrive l'architettura. In questa sede non ci dilungheremo nella spiegazione di ciascuno dei componenti raffigurati, ma utilizzeremo l'immagine per inquadrare quello che è l'elemento forse più importante per noi sviluppatori, ossia la Dalvik VM. Notiamo come essa faccia parte dell'ambiente di runtime e sia costruita sui servizi offerti dal kernel e dalle diverse librerie native; l'altro elemento importante del runtime è costituito dalle core library. Ciò che l'architettura non mette in evidenza è il ruolo di Java in tutto questo. In un'architettura Android non esiste alcuna virtual machine Java e non viene eseguito alcun bytecode Java. Il fatto che Android non fosse una specializzazione della piattaforma Java ME l'avevamo capito nell'Introduzione, per cui perché parliamo di Java? Semplicemente perché per sviluppare applicazioni per Android si utilizza (principalmente) Java. Ma allora perché non si è utilizzata la Java ME? Beh, la piattaforma Java ME purtroppo non ha avuto lo stesso successo delle sorelle Java Standard Edition e Java Enterprise Edition, non avendo mantenuto la promessa del "Write Once, Run Everywhere".

> **NOTA**
>
> Con JME intendiamo la *Java Micro Edition* le cui specifiche erano state rilasciate da Sun, ora acquisita da Oracle. Con MIDP 2.0 intendiamo invece un particolare insieme di librerie che hanno caratterizzato le applicazioni sui dispositivi di ormai qualche anno fa di costruzione principalmente Nokia. Per intenderci, è l'ambiente utilizzato per la realizzazione dei classici giochini per cellulare ormai estinti a favore delle applicazioni su smartphone.

I dispositivi mobili in grado di eseguire delle MIDlet erano moltissimi e molto diversi tra loro. Realizzare delle applicazioni che funzionassero bene e sfruttassero a dovere le caratteristiche di ciascun dispositivo si è rivelata cosa impossibile. Un dispositivo può promettere di eseguire applicazioni MIDP 2.0 allo stesso modo di un altro, ma può avere dimensioni, risoluzione e modalità di input diverse. Alcuni hanno il Bluetooth altri no, alcuni hanno la fotocamera e altri no. Per questo motivo si è portati a realizzare applicazioni specifiche per classi di dispositivi come possono essere quelli raggruppati sotto il nome di "classe 60" di Symbian.

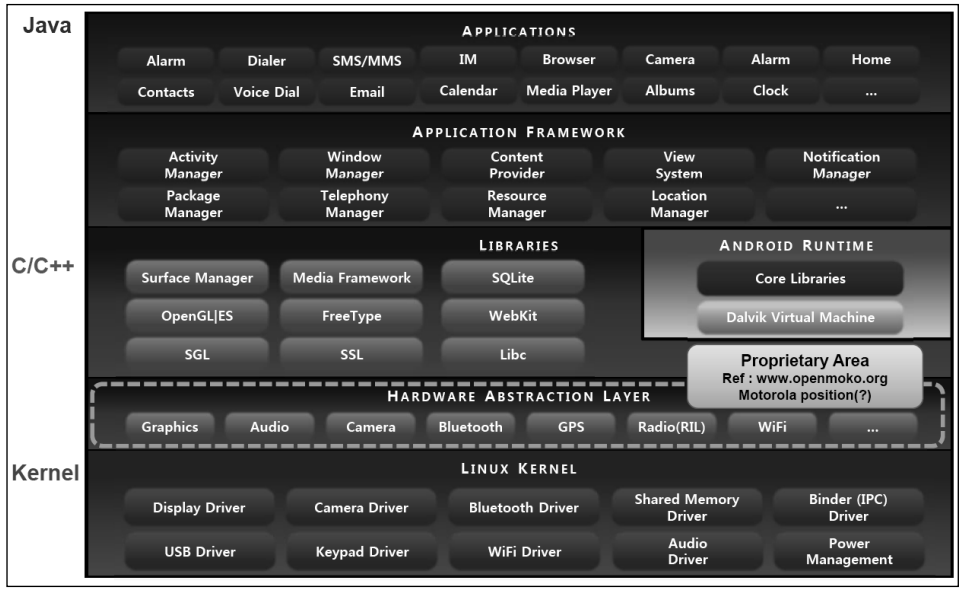

Figura 1.1 Architettura di Android (`http://www.gurubee.net/`).

Un altro aspetto molto importante di Java ME è il suo legame con la piattaforma madre Java SE, di cui è un sottoinsieme. Attraverso la classificazione in *Configuration* e *Profile* si è infatti deciso di eliminare dalla versione Java SE tutte quelle istruzioni e API non in grado di poter essere eseguite in un dispositivo con risorse limitate in termini di CPU e memoria. La Java ME non è quindi stata pensata appositamente come piattaforma mobile ma come "l'insieme delle API che funzionano anche su dispositivi mobili". L'OHA e Google hanno quindi deciso che la cosa migliore per la realizzazione di una piattaforma mobile fosse quella di creare subito qualcosa di ottimizzato e specifico senza ricadere nel classico "reinvent the wheel" per i componenti principali.
Bene, fin qui abbiamo capito che la Java ME non c'è; ma allora Java cosa c'entra? Un obiettivo non secondario della nuova piattaforma è quello di rendere relativamente semplice la creazione di applicazioni che alla fine sono quelle che ne determinano il successo. Le possibilità erano quelle di creare un nuovo linguaggio oppure di utilizzarne uno esistente ereditando una community di sviluppatori e soprattutto un insieme maturo di strumenti di sviluppo. La scelta è caduta su Java non solo per questi motivi ma anche per l'opportunità di avere a disposizione un bytecode di specifiche conosciute (e, per i maligni, anche libero da qualunque royalty per la certificazione TCK) e quindi modificabile e ottimizzabile a piacere.
Un'applicazione Android si sviluppa in Java viene compilata in un insieme di risorse e bytecode Java. Quest'ultimo viene poi trasformato in bytecode Dalvik attraverso l'applicazione di una serie di ottimizzazioni che lo rendono più compatto ed efficiente a runtime. Ecco che da un'applicazione Java si ottiene un bytecode non Java che viene eseguito in una virtual machine non Java.
Come detto non andremo a descrivere tutti i componenti dell'architettura precedente ma possiamo comunque osservare che ci sono sostanzialmente tre diverse parti:

- sistema operativo;
- librerie;
- application framework.

Il primo contiene la parte dell'architettura di competenza dei costruttori, i quali dovranno adattare il sistema operativo con il kernel Linux di riferimento e quindi realizzare i diversi driver per l'interazione con il particolare hardware. La seconda parte è quella delle librerie e ha un'importantissima caratteristica: non sono sviluppate in Java ma perlopiù in C/C++. Ebbene sì, le applicazioni Android si sviluppano (principalmente) in Java ma la maggior parte del codice che verrà eseguito sarà codice nativo in C/C++. Questo per il semplice fatto che molti dei componenti della piattaforma sono semplicemente delle interfacce Java di componenti nativi. Questa affermazione era vera soprattutto nelle prime versioni della piattaforma; dalla versione 2.2 ci si è accorti che comunque la gran parte del codice di un'applicazione faceva riferimento alla UI descritta appunto attraverso del bytecode Dalvik. Per questo motivo da questa versione è stato introdotto anche un garbage collector intelligente in grado di portare diversi benefici nella gestione della memoria. Infine, il framework applicativo è quella parte di architettura che contiene tutti componenti che andremo a utilizzare, personalizzare ed estendere per la realizzazione delle nostre applicazioni. Come vedremo nei prossimi capitoli, una delle principali caratteristiche di Android è quella di permettere la realizzazione di applicazioni attraverso gli stessi strumenti utilizzati per creare la piattaforma stessa. Potremmo creare, per esempio, la nostra applicazione di e-mail da utilizzare poi al posto di quella di Gmail installata su quasi tutti i dispositivi, oppure realizzare un'applicazione dei contatti che vada a sostitire quella presente.Viceversa le nostre applicazioni potranno utilizzare componenti già esistenti. Caso tipico è quello di un'applicazione che vuole inviare un'immagine a un determinato contatto attraverso una mail. In Android questa applicazione non sarà obbligata a sviluppare la parte di gestione dei contatti, la gallery per la scelta dell'immagine e neppure il componente per l'invio del messaggio.Attraverso i concetti fondamentali di *intent* e *intent filter* che vedremo nel Capitolo 3, capiremo come sia possibile scegliere un contatto tra quelli disponibili utilizzando l'applicazione dei contatti esistenti, selezionare un'immagine utilizzandola gallery presente e quindi inviare l'e-mail usando una delle applicazioni esistenti. Il linguaggio che andremo a utilizzare sarà comunque Java anche se, da ormai molto tempo, Android permette lo sviluppo in modo completamente nativo attraverso un proprio ambiente chiamato NDK (*Native Development Kit*), che non tratteremo in questo testo data la vastità dell'argomento.

La Dalvik Virtual Machine

La conoscenza dei dettagli che stanno dietro questa nuova virtual machine è sinceramente superflua nella maggior parte dei casi. Una necessità di questo tipo si ha, per esempio, per alcune applicazioni native che utilizzano l'NDK e che quindi hanno bisogno di ottimizzazioni a livello di bytecode. In questa sede ci interessa quindi solamente dare alcune informazioni su cosa effettivamente renda questa VM interessante. Diciamo quindi che si tratta di una VM progettata e realizzata da Dan Borstein con il principale obiettivo di essere ottimizzata per dispositivi mobili. Essa è per esempio in grado di eseguire più processi contemporaneamente attraverso una gestione ottimale delle risorse condivise.

Una delle ottimizzazioni più conosciute riguarda il fatto che si tratta di una VM *register-based*, diversamente dalla KVM che è invece *stack-based*.

NOTA

La KVM (la *K* indica che le dimensioni erano di 70 KB circa) è la principale virtual machine utilizzata dai cellulari per l'esecuzione delle ormai famose MIDlet ovvero le applicazioni per MIDP.

Attraverso un utilizzo intelligente dei registri di sistema permette una maggiore ottimizzazione della memoria in dispositivi con bassa capacità. Si tratta inoltre di una VM ideale per l'esecuzione contemporanea di più istanze sfruttando la gestione di processi, memoria e thread del sistema operativo sottostante. La Dalvik VM non esegue bytecode Java ma un qualcosa che si può ottenere da esso e che prende il nome di Dalvik bytecode. Ecco svelato l'arcano. Le applicazioni per Android si sviluppano in Java sfruttando i tool di sviluppo classici come Eclipse, NetBeans e ora Android Studio per poi trasformare il bytecode Java in Dalvik bytecode. Su un dispositivo Android non girerà alcun bytecode Java ma un bytecode le cui specifiche sono descritte dal formato DEX (*Dalvik EXecutable*). Le core library e tutte le API che avremo a disposizione nella realizzazione delle applicazioni per Android saranno quindi classi Java.

I componenti principali di Android

La caratteristica della DVM di poter eseguire diverse istanze contemporanee in modo ottimizzato non è da sottovalutare. Questo perché Android associa a ciascuna applicazione, distribuita all'interno di un file di estensione `.apk`, un singolo task a cui viene data la responsabilità di gestire più activity, ossia quelli che sono i componenti più importanti di una qualunque applicazione per questo sistema e che rappresentano quella che è una schermata. Possiamo quindi dire che gli elementi fondamentali di Android, dal punto di vista dello sviluppatore, sono i seguenti.

- activity;
- intent e intent filter;
- broadcast intent receiver;
- service:
- content provider.

Ora li descriviamo brevemente, per poi approfondirli man mano durante la realizzazione della nostra applicazione.

Activity

Potremmo definire un'*activity* come una schermata di un'applicazione di Android, ossia un qualcosa che visualizza delle informazioni o permette l'inserimento di dati da parte dell'utente attraverso la renderizzazione di quelle che chiameremo *view*. Un'applicazione sarà perlopù costituita da più activity ciascuna delle quali verrà eseguita da un proprio processo all'interno di uno o più *task*. Il compito degli sviluppatori sarà quello di creare le diverse activity attraverso la descrizione delle view e delle modalità con cui le stesse si passano le informazioni. 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 perlopù al sistema che, in base alla necessità, può decidere di terminarne uno o più. In quel caso si dovranno adottare i giusti accorgimenti per non incorrere in perdita di informazioni. Da notare come attraverso il plug-in fornito da Google sia possibile definire le view in modo dichiarativo attraverso alcuni file XML di configurazione che prendono il nome di *layout*.

Intent e intent filter

Come abbiamo accennato, l'architettura di Android è ottimizzata in modo da permettere uno sfruttamento migliore delle risorse disponibili. Per raggiungere questo scopo si è pensato di "riciclare" quelle attività che svolgono operazioni comuni a più applicazioni. Per riprendere l'esempio precedente, consideriamo la semplice selezione di un contatto dalla rubrica. Il fatto che ciascuna applicazione possa gestire il modo con cui un'azione avviene ha il vantaggio di permettere una maggiore personalizzazione delle applicazioni, ma ha il grosso svantaggio di fornire diverse modalità, spesso pure simili tra di loro, per eseguire una stessa operazione. Un utente che utilizza un dispositivo si aspetta di compiere la stessa operazione sempre nelle stesso modo in modo indipendente dall'applicazione. Ecco che 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". A questo punto serve un meccanismo che permetta di associare tale intent a un'activity per la sua esecuzione. Ciascuna activity può dichiarare l'insieme degli intent che la stessa è in grado di esaudire attraverso quelli che si chiamano *intent filter*. Se un'activity ha tra i propri intent filter quello relativo alla scelta di un contatto dalla rubrica, quando tale intent viene richiesto essa verrà visualizzata per permettere all'utente di effettuare l'operazione voluta. Tale attività sarà la stessa per ogni applicazione senza che occorra definirne una propria. Questo utilizzo degli intent riguarda la comunicazione tra più activity.

Broadcast intent receiver

Ormai gli smartphone e i tablet ci seguono dovunque e sanno tutto di noi. Sanno dove andiamo, come ci andiamo, e sono in grado di rispondere a sollecitazioni di varia natura attraverso la percezione di informazioni attraverso un numero sempre maggiore di sensori. In un ambiente come quello Android serviva quindi un componente in grado di reagire a determinati eventi e quindi attivare un'applicazione, visualizzare una notifica, iniziare a vibrare o altro ancora. Questo tipo di 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*. Gli eventi possono essere generati dalle applicazioni o direttamente dal dispositivo. Parliamo della ricezione di una telefonata, di un SMS, del segnale di batteria scarica, della disponibilità della rete prima assente e altro ancora.

Service

In precedenza abbiamo associato le activity a delle schermate. Nel caso in cui si rendessero necessarie delle funzionalità *long running* (di esecuzione prolungata) non direttamente legate ad aspetti visuali, la piattaforma ci fornisce quelli che si chiamano *service*. Parliamo

di processi con la responsabilità di riprodurre file multimediali, leggere o scrivere informazioni attraverso la rete o attraverso le informazioni dell'eventuale GPS integrato. 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 visualizzato nel display, e quindi con ciò da cui l'utente in quel momento sta interagendo.

Content provider

Dall'immagine dell'architettura notiamo come Android metta a disposizione un SQLite per la gestione della persistenza di informazioni comunque limitate, per default, a ciascuna singola applicazione. Se si intende fornire un insieme di dati per più applicazioni, che ricordiamo sono in esecuzione in processi diversi, è di fondamentale importanza il concetto di *content provider*. 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 content provider come a una specie di DA, 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.

Cosa rende Android particolare

Dopo aver descritto i componenti principali dell'architettura di Android vediamo di descrivere alcuni degli aspetti della piattaforma che lo rendono diverso dalle tecnologie o architetture alternative.

Definizione dichiarativa della UI

Attraverso l'utilizzo di opportuni documenti XML, facilmente gestibili con il plug-in messo a disposizione da Google, è possibile creare in maniera dichiarativa l'interfaccia grafica delle applicazioni. Lo stesso plug-in permette di avere, in ogni momento, una preview del risultato dell'interfaccia realizzata. L'utilizzo di questo approccio è stato un passo obbligato legato alla presenza di innumerevoli versioni dei dispositivi Android.

Ciascuna applicazione può essere consumer e provider di informazioni

Utilizzando il meccanismo degli intent e dei content provider si può fare in modo che le diverse applicazioni collaborino tra loro per la gestione dei contenuti secondo il paradigma Web 2.0. Si tratta quindi di un vero e proprio ambiente all'interno del quale più applicazioni possono convivere con un utilizzo ragionato delle risorse. A tale proposito possiamo pensare ad Android come una vera e propria piattaforma che offre una serie di strumenti utilizzabili in modo relativamente semplice all'interno delle nostre applicazioni. Pensiamo alla possibilità di accedere ai servizi di Google Maps per la realizzazione di applicazioni location-based, o alla semplicità con cui si può interagire con i servizi di telefonia o di gestione degli SMS o con lo stesso browser integrato. Lo sviluppatore si concentrerà su quelli che sono i servizi di business senza preoccuparsi della creazione di strumenti a supporto, che potremo comunque realizzare avendo a disposizione un intero stack.

Il codice sorgente è open source

Android non solo utilizza tecnologie open source ma il suo stesso codice è ora disponibile sotto licenza Apache 2.0.

È basato sulle API della Java SE ma non esegue bytecode Java

Per quello che riguarda le API Java disponibili, queste sono relative alla piattaforma Java SE (e non a quella Java ME) con un numero di sviluppatori sicuramente superiore, che quindi può iniziare a sviluppare applicazioni Android con una curva di apprendimento molto breve. Notiamo come l'analogia con la Java SE sia stata fatta a livello di API e non di implementazione. L'applicazione in esecuzione sul dispositivo Android viene eseguita all'interno di una VM che è strettamente legata alle risorse del sistema operativo Linux. Questo significa che l'applicazione pseudo-Java è praticamente nativa rispetto al dispositivo con notevoli vantaggi dal punto di vista delle prestazioni. In un dispositivo Android non viene eseguito bytecode Java e non esiste alcuna VM Java.
Quelle descritte sono tutte caratteristiche che fanno di Android una piattaforma di grande interesse che impareremo a padroneggiare per la realizzazione di applicazioni molto interessanti.

Le applicazioni di Android

Come detto, il successo di una piattaforma come Android verrà deciso dalle applicazioni disponibili. A tale proposito la politica di Google è molto chiara e prevede la possibilità di utilizzo di un ambiente di sviluppo e di un emulatore per il test delle applicazioni. Attraverso quello che si chiama Google Play chiunque, dopo una spesa di 25$, può vendere (o regalare) le proprie applicazioni. È una politica per certi versi opposta a quella di iPhone, dove per sviluppare le applicazioni è necessario registrarsi alla comunità di sviluppatori e utilizzare anche strumenti e canali ben definiti. Non ci resta quindi che dare sfogo alla nostra fantasia e realizzare non solo i classici giochi ma soprattutto applicazioni che sfruttano le potenzialità delle mappe di Google e la possibilità di interagire con le funzionalità del telefono. Pensiamo non solo al fatto di poter cercare i ristoranti o altri punti di interesse vicini alla nostra attuale posizione, ma anche di determinare il tragitto per arrivarvi o iniziare una chiamata per la prenotazione. Sebbene non si tratti certo di un'applicazione originale, con Android la novità consiste nella semplicità con cui la stessa può essere realizzata e nel fatto che essa potrà veramente essere fruita da tutti i dispositivi abilitati, che cominceranno presto a essere prodotti da varie aziende.

Introduzione a Java

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 che sono gli aspetti del linguaggio che ci saranno più utili. Il lettore già a conoscenza del linguaggio potrà tranquillamente passare al capitolo successivo, 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 capitolo successivo.

Iniziamo allora dalla celeberrima applicazione `Hello World`, leggermente modificata, di cui riportiamo il codice nella sua interezza:

```
package uk.co.massimocarli.javacourse.hello;

import java.util.Date;

/**
 * La classica applicazione Hello World
 */
public class HelloWorld {

    /*
     * Il messaggio da stampare
     */
    private static final String MESSAGE = "Hello World!";

    /**
     * Questo è il metodo principale dell'applicazione
     * @param args: array con i parametri dell'applicazione
     */
    public static void main(String[] args) {
        final Date now  = new Date();
        // Stampa il messaggio Hello World!
        System.out.println(MESSAGE + " " + now);
    }

}
```

Dalla prima istruzione che inizia con la parola chiave `package` seguita da un identificativo composto da tutte parole minuscole divise dal punto. Un package rappresenta quindi un modo per raggruppare definizioni che fanno riferimento a cose comuni. Per esempio, la piattaforma fornisce il package `java.net`, che raccoglie tutte le classi e gli strumenti relativi al networking, alle operazioni di I/O e così via. In Java non possiamo creare definizioni in package che iniziano per `java` o `javax` e vedremo che una limitazione simile esiste anche per le definizioni dei package che iniziano per `android`.

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` o `annotation`. Per semplificare di seguito faremo riferimento alle classi ricordandoci però che lo stesso varrà per queste due interfacce.

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 lo stesso inizi per 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 applicazione che la stessa vorrà produrre saranno del tipo:

```
co.uk.miazienda.android
```

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

```
nome1.nome2
```

e il package

```
nome1.nome2.nome3
```

in quanto si tratta semplicemente di due package di nomi diversi. 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.massimocarli.javacourse.hello.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 o l'interfaccia `List` del package `java.util` per la gestione della famosa struttura dati.
Abbiamo quindi 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 tutte le classi Java in modo diretto o indiretto, la classe `String`, tutti i wrapper (`Integer`, `Boolean`…) e così via. Nel caso in cui 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 possiamo semplicemente utilizzare lo stesso tipo di istruzione oppure scrivere

```
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` ("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.
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 appunto di definire una class, 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 {
    - - -
}
```

e la visibilità sarebbe stata ristretta solamente alle classi dello stesso package. È importante quindi 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`.

> **ATTENZIONE**
>
> 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 che può essere applicata solamente ad attributi, come in questo caso, e a metodi. 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 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. Si parla di attributi di classe in quanto vengono creati nel momento in cui il bytecode viene letto e interpretato dal `ClassLoader`. Proprio per questo motivo 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, l'accesso a esso avverrebbe attraverso questa sintassi:

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

```
HelloWorld.MESSAGE
```

Il modificatore `static` applicato ai metodi ha un significato analogo per cui permette la definizione di metodi che non dipendono dallo stato delle istanze di una classe e che possono essere invocati come nell'esempio precedente. Si tratta principalmente di metodi di utilità che hanno nei propri parametri tutte le informazioni che servono per assolvere al proprio compito. Vedremo tra poco cosa comporterà questa osservazione con il metodo `main()`. Una top class non può essere definita come `static`, a differenza di quelle interne che vedremo più avanti.
Il terzo modificatore è molto importante e si chiama `final`. Può essere applicato sia alle classi sia agli attributi e metodi ma 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 appunto 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 essere valorizzato solamente una volta e prima che venga invocato il costruttore della corrispondente classe.
Quella precedente è quindi 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 `String`. È importante sottolineare come questo debba essere il metodo di avvio dell'applicazione; ogni eventuale differenza renderebbe l'applicazione non avviabile.
Se volessimo eseguire la nostra applicazione da riga di comando utilizzeremmo la prossima istruzione, dove notiamo che la classe è specificata in modo comprensivo del relativo package:

```
java  uk.co.massimocarli.javacourse.hello.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 di ritorno e si deve chiamare `main` con un array `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 poi per la visualizzazione del nostro messaggio.
È buona norma utilizzare il modificare `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 2013
```

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 una stringa si ha l'invocazione del 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 azienda. I modificatori non solo di visibilità di Java sono diversi, ma quelli elencati sono sicuramente i più importanti. Avremo comunque occasione di ulteriori approfondimenti durante la realizzazione 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 ad oggetti più importanti, che sappiamo essere:

- incapsulamento;
- ereditarietà;
- polimorfismo.

Sono tra loro legati e sono stati introdotti con l'unico scopo di diminuire l'impatto delle modifiche nel codice di un'applicazione. Sappiamo infatti che il costo di una modifica cresce in modo esponenziale man mano che si passa da una fase di analisi alla fase di progettazione e sviluppo. In realtà il nemico vero 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 come lo stesso assolve alle proprie responsabilità debbano rimanere nascosti e quindi non visibili agli oggetti con cui lo stesso 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 nello spazio. Una prima implementazione di questa classe potrebbe essere questa:

```
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 uti-

lizzano 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:

```
public class Point2D {

    private double x;

    private double y;

    public EncapsulatedPoint2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

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

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

}
```

In questo caso gli utenti interagiscono con il nostro punto utilizzando i metodi `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 Point2D {

    private double r;

    private double theta;

    public Point2D(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 double getY() {
        return r * Math.sin(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 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 come un oggetto esegue le proprie operazioni in modo 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. 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 la moltiplicazione di due interi. Una prima implementazione potrebbe essere questa:

```
public class Calculator {

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

}
```

la quale prevede appunto un metodo `multiply()` che moltiplica due valori interi sommando il secondo tante volte quanto indicato dal primo. Supponiamo di utilizzare questa versione per poi accorgerci che in alcuni casi non funziona. Pensiamo per esempio al caso in cui i parametri 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 utilizzava il metodo `multiply()` sull'oggetto `Calculator` prima 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 è cosa un oggetto fa e non come questo avviene. Cosa un oggetto è in grado di fare può essere descritto attraverso un'interfaccia definita nel seguente modo:

```
public interface Calculator {

    int multiply(int a, int b);

}
```

Si tratta quindi 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" questa interfaccia. L'implementazione precedente può quindi essere definita nel seguente modo:

```
public class MyCalculator implements Calculator {

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

}
```

Ma qual è il vantaggio che si ha nel definire un'interfaccia e quindi un insieme di possibili implementazioni? Si tratta del concetto forse più importante della programmazione ad oggetti: il *polimorfismo*. Ogni oggetto interessato a utilizzare un `Calculator` 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 nemico è la dipendenza, che è tanto inferiore quanto meno i diversi oggetti conoscono uno dell'altro in relazione alla propria 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 `Calculator` e quindi il codice precedente può diventare:

```
Calculator 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 del codice del tipo:

```
private Calculator mCalculator;

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

- - -
int res = mCalculator.multiply(10,20);
```

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

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

Attraverso un riferimento di tipo `Calculator` 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 `Calculator` 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 quella vista per le interfacce di *interface inheritance*) rappresenta infatti il vincolo più forte di dipendenza. Come vedremo in questo libro, la realizzazione di un'applicazione Android prevede la creazione di alcuni componenti come specializzazione 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 come per esempio iOS. Una delle regole principali della programmazione ad 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 che utilizzarlo attraverso un riferimento di tipo `Calculator` come fatto nell'esempio. L'ereditarietà in Java presenta inoltre un problema relativo al fatto che è singola. Ogni classe in 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 classi del framework. Andando sul concreto, supponiamo di voler utilizzare la *Compatibility Library,* una libreria messa a disposizione da Google per poter usare i fragment anche in versioni precedenti la 3.0, nella quale sono stati ufficialmente introdotti.

NOTA

I fragment verranno affrontati nel Capitolo 4 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 nelle nostra gerarchia fino alla classe che estende `activity` sostituendola poi con `FragmentActivity`. Il problema si ha qualora si decidesse di utilizzare `ActionBarSherlock`, una libreria che permette di utilizzare l'`ActionBar` anche in versioni della piattaforma precedenti la 3.0. Anche questo framework prevede la necessità di estendere una specializzazione della classe `Activity`. Se poi decidessimo di utilizzare anche un framework che si chiama `Roboguice`? Diciamo quindi che questa decisione di basare tutto sull'ereditarietà porta a qualche complicazione che dovremo affrontare di volta in volta.

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 quelle che sono 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 quindi ne ricevono in qualche modo la notifica. Serve un meccanismo affinché la sorgente dell'evento non sappia a priori chi sono quelli che chiameremo *listener*, i quali potrebbero essere istanze di classi qualunque. 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 una classe di nome `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 passato 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 detto, dovranno essere visti dalla sorgente come tutti dello stesso tipo da cui la necessità della creazione di 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 loro elaborazione delle informazioni ricevute. Serve ora un meccanismo per informare la sorgente dell'interesse verso l'evento. Le specifiche JavaBeans prevedono la definizione di metodi del tipo:

```
public void addTickListener(TicListener listener);
public void removeTickListener(TicListener listener);
```

che nel caso Android, ma anche 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.

Il metodo `remove` è stato eliminato; sarà quindi sufficiente impostare il valore `null` come listener attraverso il precedente metodo `set`.
Durante lo sviluppo della nostra applicazione avremo modo di vedere quali eventi gestire e come. Per fare questo 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 feature 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 la creazione di determinate classi con visibilità limitata a quella di una classe iniziale, che condividesse con essa alcuni dati o che semplicemente fosse legata alla prima 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 feature 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

package 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 colleghiamo a quanto detto in relazione alla gestione degli eventi e definiamo la seguente classe:

```
public class EventSource {

    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;
        }

    }

    public interface EventListener {

        void eventTriggered(MyEvent event);

    }

    private EventListener mListener;

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

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

}
```

La nostra classe si chiama quindi `EventSource` e rappresenta la sorgente di un evento che abbiamo descritto attraverso una classe interna statica `MyEvent`. Notiamo innanzitutto come si tratta di una classe statica, che quindi non potrebbe accedere a membri non statici della classe esterna. Se provassimo a utilizzare, per esempio, la variabile `mListener`, noteremmo 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 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 giusto 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 uno privato che la sorgente utilizzerà per la notifica dell'evento all'eventuale listener. In questa fase è importante notare come le classi in gioco siamo, 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:

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

sempre nel caso in cui la stessa fosse comunque visibile (cosa che nel nostro esempio precedente non avviene). 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 {
    public void eventTriggered(MyEvent event){
        - - -
    }
}
```

e non semplicemente

```
public class MyListener implements EventListener {
    public void eventTriggered(MyEvent event){
        - - -
    }
}
```

Questo a meno di non 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 trascurare a favore della leggibilità del codice.
Prima di passare a un altro tipo di classe interna facciamo una piccola ma importante osservazione. Mentre la definizione dell'interfaccia è la seguente:

```
public interface EventListener {

    void eventTriggered(MyEvent event);

}
```

le implementazioni sono del tipo:

```
public class MyListener implements EventSource.EventListener {
    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 EventSource.EventListener {
    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 String mName;

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

    public class Printer {

        public void printName(){
            System.out.println("The use 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 compila facilmente. La domanda che ci poniamo riguarda il come si possa 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) {
    UserData pippo = new UserData("pippo");
    UserData.Printer p1 = pippo.new Printer();
    UserData pluto = new UserData("pluto");
    UserData.Printer p2 = pluto.new Printer();
}
```

La sintassi da utilizzare è quindi la seguente:

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

A dire il vero in Android non utilizzeremo quasi mai questa sintassi in quanto creeremo le istanze delle classi interne direttamente all'interno della classe esterna. Un tipico esempio che vedremo nel dettaglio nel Capitolo 3 è questo:

```
public class SplashActivity extends Activity {

    - - -

    private  MyHandler mHandler;

    private class MyHandler extends Handler {

        @Override
        public void handleMessage(Message msg) {
          // Fai qualcosa
        }

    };

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_splash);
      mHandler = new MyHandler();
    }

    @Override
    protected void onStart() {
        super.onStart();
        mStartTime = SystemClock.uptimeMillis();
        final Message goAheadMessage = mHandler.obtainMessage(GO_AHEAD_WHAT);
        mHandler.sendMessageAtTime(goAheadMessage, mStartTime + MAX_WAIT_INTERVAL);
    }

}
```

dove creeremo una classe interna di nome `MyHandler` che poi istanzieremo e utilizzeremo direttamente dall'interno della nostra classe. La creazione delle istanze dei componenti della piattaforma è infatti responsabilità della piattaforma.

Classi locali

Proseguiamo la descrizione delle classi interne con quelle che si chiamano *locali* e che, analogamente a come 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 mSource1;

    private EventSource mSource2;
```

```
    public LocalClassTest() {
        mSource1 = new EventSource();
        mSource2 = new EventSource();
    }

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

            @Override
            public void eventTriggered(MyEvent event) {
                // Gestione evento
                System.out.println("Event with num " + num);
            }

        }
        mSource1.setEventListener(new Local());
        mSource2.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 successivamente 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 EventSource.EventListener externalise() {
    final int num = 0;
    // Definiamo una classe locale
    class Local implements EventSource.EventListener {

        @Override
        public void eventTriggered(MyEvent event) {
            // Gestione evento
            System.out.println("Event with num " + num);
        }

    }
    return new Local();
}
```

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

Classi anonime

Eccoci finalmente al tipo di classi interne più utili, ovvero a quelle *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  EventSource.EventListener {
    @Override
    public void eventTriggered(MyEvent event) {
        System.out.println("Event with num " + num);
    }
}
source.setEventListener(new MyListener());
```

A questo punto ci domandiamo se valga la pena definire una classe, e quindi una sua istanza, quando quello che a noi 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 MyListener(){
    @Override
    public void eventTriggered(MyEvent event) {
        System.out.println("Event with num " + num);
    }
});
```

Notiamo come si possa creare "on the fly" 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
};
```

dove 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'ultimo argomento che ci accingiamo a trattare 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 le stesse gestiscono. Anche qui ci aiutiamo con qualche esempio che utilizza quelle che si chiamano *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 in modo sequenziale che può essere implementata in modi diversi. Pensiamo per esempio a un'implementazione che utilizza un array descritto dalla classe `ArrayList` o quella che prevede l'utilizzo di una lista concatenata descritta dalla classe `LinkedList`. In ogni caso, in una lista possiamo aggiungere elementi e quindi accedere agli stessi 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);
```

le quali permettono la creazione di una lista implementata come lista concatenata, a cui aggiungiamo tre oggetti. Quest'ultima affermazione potrebbe trarre in inganno se non si conoscesse un'altra importante feature 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++) {
    Integer item = (Integer) list.get(i);
    System.out.println("Item :" + 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 compila 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.massimocarli.javacourse.oo.generics.GenericTest.
main(GenericTest.java:23)
```

Abbiamo quindi 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:

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

In questo caso un'istruzione del tipo

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

porterebbe a un errore in compilazione in quanto non si può inserire una `String` in una lista di `Integer`. I vantaggi non si hanno poi 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 prossima è perfettamente legale:

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

I generics sono comunque molto di più. Supponiamo infatti 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 stampa a video. Il primo tentativo del programmatore inesperto sarebbe questo:

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

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

ovvero la realizzazione di 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 feature, anche quella dei generics è legata al compilatore, che comunque crea una `List` di oggetti in entrambi i casi. Il workaround più comunque sarebbe quello di cambiare il nome dei metodi e quindi utilizzarli nel seguente modo:

```
public class GenericTest2 {

    public static void main(String[] args) {
        List<Integer> list = new LinkedList<Integer>();
        list.add(1);
        list.add(2);
        list.add(3);
        printInteger(list);

        List<String> list2 = new LinkedList<String>();
```

```
        list2.add("one");
        list2.add("two");
        list2.add("three");
        printString(list2);

    }

    public static void printInteger(List<Integer> list) {
        for (Integer item: list) {
            System.out.println(item);
        }
    }

    public static void printString(List<String> list) {
        for (String item: list) {
            System.out.println(item);
        }
    }

}
```

In questo caso il codice compilerebbe e verrebbe 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 quello che si chiama *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(item);
    }
}
```

il quale utilizza una notazione all'apparenza strana che ci permette di definire quella che è una `List` di *unknown*. Attraverso un riferimento di tipo `List<?>` possiamo quindi 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 descrive una specializzazione della prima. In sintesi, un cane è un animale e quindi un `Dog is an 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();
```

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

```
a.bark()
```

in quanto sebbene un cane sia in grado di abbaiare (`bark`) e l'oggetto referenziato da `a` sia effettivamente un cane, noi lo stiamo osservando 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:

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

ovvero se un array di `Dog` è un array di `Animal`. 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

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

è perfettamente valida e infatti compila e viene pure eseguita correttamente. Consideriamo ora però questa istruzione

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

dove anche `Cat` è una classe che estende `Animal` in quanto anche un gatto è un animale. Anche qui il codice compila in quanto `a[1]` è un riferimento a un oggetto di tipo `Animal` che nell'istruzione precedente viene valorizzato 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.massimocarli.
javacourse.oo.generics.Cat at uk.co.massimocarli.javacourse.oo.generics.GenericTest3.
main(GenericTest3.java:13)
```

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 compila e, in caso affermativo, se viene eseguita correttamente:

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

Ci chiediamo se una `List` di `Animal` è 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 però ora è riconosciuto dal compilatore, che ci notifica un errore che quindi 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>();
```

e quindi le seguenti istruzioni diventano perfettamente lecite:

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

con un prezzo che è quello di non poter inserire ma solamente estrarre. Questo perché, per quanto detto, se potessimo inserire avremmo la capacità 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 di che tipo sarà l'oggetto estratto? Anche qui la risposta è un salomonico "non lo sappiamo", ma non siamo completamente ignoranti. 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(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`. Da un lato questo ci permette 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 fatto nel nostro *enhanced for*. In questo caso possiamo anche inserire qualcosa? Purtroppo ancora no, perché sappiamo 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 l'inserimento di oggetti all'interno di 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

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

permette di referenziare una lista di `Dog`, oppure di `Animal` oppure di `Object`. Prendendo sempre questo ultimo esempio, in una lista come questa possiamo anche inserire, 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 un metodo di ordinamento di oggetti di tipo `T` contenuti in una lista attraverso questo tipo di `Comparator`. 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 tra essi. 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 il `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 final T mData;

    public Holder(T data){
        this.mData = data;
    }

    public T getData() {
        return mData;
    }

}
```

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 delle righe del codice di questo tipo:

```
public static void main(String[] args) {
    Holder<String> hString = new Holder<String>("String");
    String str = hString.getData();
    Holder<Integer> hInteger = new Holder<Integer>(10);
    Integer val = hInteger.getData();
}
```

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 `getData()`. L'ultima osservazione riguarda quella che si chiama *type inference* e 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, ritorna 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 ritornata dal metodo che chiamiamo `getView()` sia quello della variabile a cui viene assegnato. Osserviamo la seguente implementazione di un metodo statico che supponiamo essere della classe UI:

```
public static <T extends View> T findViewById(Activity act, int viewId) {
    // Otteniamo il ViewGroup dell'activity
    View containerView = act.getWindow().getDecorView();
    return findViewById(containerView, viewId);
}
```

Attraverso l'utilizzo di 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 ritornato sia effettivamente del tipo voluto. Lo stesso problema si aveva comunque anche nel caso precedente, ma così si evita il cast.

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 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.