

Introduzione ad Android

La fine del precedente millennio è stata sicuramente caratterizzata da Internet: una vera e propria rivoluzione non solo tecnologica, ma soprattutto culturale. Il fatto di permettere a chiunque di pubblicare informazioni accessibili da una qualunque parte del mondo ha permesso una maggiore divulgazione delle informazioni e una migliore distribuzione e condivisione della conoscenza.

Dal punto di vista informatico, Internet ha introdotto un nuovo paradigma per lo sviluppo delle applicazioni, accompagnato da nuove tecnologie e scelte di architettura. Da programmi desktop in esecuzione sui diversi PC siamo passati ad applicazioni web accessibili da una qualsiasi parte del mondo attraverso quello che si definisce uno “thin-client”, categoria di cui il browser è il principale esempio.

Nonostante non sia passato molto tempo dall'avvento di Internet, oggi stiamo vivendo una nuova rivoluzione: quella dei dispositivi mobili. Quelle che prima erano informazioni e applicazioni raggiungibili ed eseguibili attraverso un qualunque PC, sono ora accessibili tramite dispositivi sempre più potenti, che hanno la fondamentale caratteristica di essere mobili e di dimensioni sempre più adatte alle funzionalità che si intende mettere a disposizione dell'utente.

Ciò che finora è stato considerato come un Personal Computer, sta diventando sempre più personale; non solo perché utilizzato da un unico utente, ma proprio perché ci accompagna ovunque, in ogni momento della giornata. Ciò che prima stava sulla nostra scrivania, ora ci può stare nel taschino. Stiamo parlando di dispositivi mobili, di quelli che prima chiamavamo in modo riduttivo con il termine di cellulare,

In questo capitolo

- **Che cos'è Android**
- **La Dalvik Virtual Machine**
- **L'architettura di Android**

ma che ora stanno diventando veri e propri PC portatili, dove la funzione di telefono, sebbene fondamentale, è solo una di quelle disponibili. Agli *smartphone*, ovvero telefoni con funzionalità avanzate, si sono poi aggiunti i *tablet*, sulla scia dell'iPad Apple. Come vedremo nei prossimi capitoli, definire un *tablet* solamente come uno *smartphone* con il display più grande è alquanto riduttivo. Si tratta invece di macchine con caratteristiche hardware diverse, che le rendono più idonee, per esempio, alla lettura di documenti, alla visualizzazione di filmati o alla gestione di applicazioni in ambiti specialistici, per esempio ospedali, magazzini e così via.

Per gli sviluppatori si sta aprendo quindi un nuovo orizzonte: quello della creazione e dello sviluppo di applicazioni che sfruttino le caratteristiche di questi dispositivi, definiti, ormai sempre meno, "a risorse limitate". Ovviamente un PC di media potenza ha caratteristiche software e hardware (quantità di memoria, potenza di CPU e alimentazione) superiori a quelle di un qualunque dispositivo mobile attuale, anche se da tempo sono comparsi tablet e smartphone con microprocessori *dual core*, cosa impensabile fino a qualche anno fa. Si stima che la potenza di calcolo di un cellulare di nuova generazione sia paragonabile a quella di un PC di 8 o 10 anni fa; ma la distanza si riduce anno dopo anno.

In questo contesto i principali costruttori di cellulari hanno messo a disposizione degli sviluppatori i propri sistemi operativi, ciascuno con il proprio ambiente di sviluppo, i propri tool e il proprio linguaggio di programmazione. Nessuno di questi, però, si è affermato come standard. Per esempio, per realizzare un'applicazione nativa (non web) per iPhone è necessario disporre del sistema operativo Mac OS X, su cui si basa l'iPhone, e conoscere il linguaggio Objective-C (un'estensione del linguaggio C, con caratteristiche Object Oriented) giunto ora alla versione 2.0. Per lo sviluppo di un'applicazione per un dispositivo Nokia, basata sul sistema operativo Symbian, è necessario utilizzare come linguaggio un dialetto del C++. Dopo l'accordo con Microsoft, sviluppare per Nokia significherà anche utilizzare gli strumenti per la creazione di applicazioni per Windows Phone 7, che avrà sicuramente un ruolo importante nei prossimi anni. Non possiamo ovviamente trascurare la piattaforma J2ME (Java 2 Mobile Edition), per la quale faremo ulteriori considerazioni nel presente capitolo, anche alla luce del suo utilizzo per lo sviluppo di applicazioni per Blackberry (RIM). Oltre questi, vi sono vari sistemi operativi proprietari, la cui conoscenza è spesso limitata ai soli vendor.

Ciò che si vuole sottolineare è, comunque, la varietà di ambienti e tecnologie che uno sviluppatore deve conoscere per poter realizzare un'applicazione per un particolare dispositivo mobile. A seconda del tipo di sistema operativo, è necessario acquisire la conoscenza di un ambiente, di una piattaforma e di un linguaggio. Esiste quindi la necessità di una standardizzazione, verso cui si sono diretti Google e la Open Handset Alliance con la creazione di Android, argomento di questo libro.

Che cos'è Android

Android non è un linguaggio di programmazione, ma un vero e proprio insieme di strumenti e librerie per la realizzazione di applicazioni mobili. Ha come obiettivo quello di fornire tutto ciò che un operatore, un vendor di dispositivi o uno sviluppatore ha bisogno per raggiungere i propri obiettivi. A differenza di alcuni degli ambienti descritti in precedenza, Android ha la fondamentale caratteristica di essere *open*, dove il termine assume diversi significati. È open in quanto utilizza, come vedremo meglio più avanti,

tecnologie open; prima fra tutte il kernel di Linux, nella versione 2.6. È open in quanto le librerie e le API che sono state utilizzate per la sua realizzazione sono esattamente le stesse che andremo a usare per creare le nostre applicazioni. Quasi tutti i componenti di Android potranno essere rimpiazzati dai nostri, così da non avere limiti nella personalizzazione dell'ambiente, se non quelli legati ad aspetti di sicurezza nell'utilizzo, per esempio, delle funzionalità del telefono. Android è open in quanto il suo codice è open-source e consultabile da chiunque intenda contribuire a migliorarlo, voglia creare una parte della documentazione o, semplicemente, voglia scoprirne il funzionamento. La licenza scelta dalla Open Handset Alliance (<http://www.openhandsetalliance.com>) è la Open Source Apache License 2.0, la quale permette ai diversi vendor di costruire su Android le proprie estensioni anche proprietarie senza legami che ne potrebbero limitare l'utilizzo. Questo significa che un produttore non deve pagare alcuna royalty per adottare Android sui propri dispositivi.

Una storia ormai lunga

Come ogni altra tecnologia, anche Android nasce da un'esigenza: quella di fornire una piattaforma aperta, e per quanto possibile standard, per la realizzazione di applicazioni mobili. Google non ha realizzato Android partendo da zero, ma ha acquisito nel 2005 la società Android Inc. con i principali realizzatori che hanno poi fatto parte del team di progettazione della piattaforma in Google. Nel 2007 le principali aziende nel mondo della telefonia hanno dato origine alla Open Handset Alliance (OHA). È curioso come le aziende che fanno parte della OHA, riguardino tutto lo stack gestito da Android. Oltre a Google, troviamo, infatti, produttori di dispositivi come Motorola, Sprint-Nextel, HTC, Samsung, Sony-Ericsson e Toshiba; operatori mobili come Vodafone, T-Mobile; infine costruttori di componenti come Intel e Texas Instruments. L'obiettivo è quindi quello di creare una piattaforma open in grado di tenere il passo del mercato senza il peso di royalty che ne possano frenare lo sviluppo.

Nel 2007 è uscita finalmente la prima versione del Software Development Kit (SDK), in modo che gli sviluppatori iniziassero a tastare con mano la nuova piattaforma, realizzando le prime applicazioni sperimentali, che dal 2008 poterono anche essere provate sul primo dispositivo reale ovvero il G1 di T-Mobile. Un passo fondamentale nella storia di Android è avvenuto nell'ottobre del 2008 quando è stato rilasciato il codice sorgente open-source con la licenza Apache ed è stata annunciata la release candidate dell'SDK 1.0. Verso la fine del 2008, Google ha dato la possibilità agli sviluppatori di alcuni paesi (fra i quali non l'Italia) di acquistare al costo di circa 400\$ un telefono, il Dev Phone 1, per sperimentare l'uso delle applicazioni senza alcun vincolo con un operatore mobile. La versione 1.0 è stata affinata fino al rilascio della versione 1.1 nel dicembre del 2008. Si è trattato di una versione di bug fixing, senza grosse novità rispetto alla precedente, se non l'eliminazione, per motivi di sicurezza, delle API che permettevano l'utilizzo dei servizi di GTalk e di altre che esamineremo in dettaglio più avanti.

Una delle limitazioni della versione 1.1 era sicuramente quella che obbligava i dispositivi ad avere una tastiera fisica.

Nella versione 1.5 dell'SDK (soprannominata *Cupcake*) rilasciata nell'aprile del 2009, una delle principali novità è stata l'introduzione della gestione della tastiera virtuale, che quindi non imponeva ai costruttori di hardware il vincolo della realizzazione di una tastiera fisica. Altre importanti novità della versione 1.5 sono state sicuramente la possibilità

di aggiungere widget alla home del dispositivo e le live folder che, come vedremo in dettaglio nel Capitolo 11, permettono l'esposizione attraverso la home di informazioni gestite da un *Content Provider*.



Figura 1.1 Il Dev Phone 1.

Il 16 settembre 2009 è stato poi rilasciato l'SDK nella versione 1.6, con importanti novità sia a livello utente, sia a livello di API. Forse la più importante riguarda la possibilità di integrare le applicazioni con quella che si chiama *Quick Search Box*, un'area di testo nella home del dispositivo, all'interno della quale ricercare informazioni di un qualunque tipo: dal meteo, al numero di telefono di un contatto, al ristorante più vicino. Questo è stato reso possibile attraverso una riprogettazione del framework di ricerca. Altra importante novità riguarda l'integrazione di Pico, ovvero di un potente sintetizzatore vocale con accenti che riflettono le principali lingue, tra cui anche l'italiano. Importante anche l'aggiunta delle API per il riconoscimento vocale. Tra le altre cose che vedremo nei prossimi capitoli è stato aggiunto il supporto al CDMA nello stack relativo al telefono e il primo supporto verso display con una più ampia gamma di dimensioni e risoluzioni. Da non trascurare, inoltre, un framework di gestione di quelle che vengono chiamate *gesture* e che offrono modi alternativi per interagire con il dispositivo.

Il 28 ottobre 2009, dopo poche settimane dal rilascio della 1.6, è stato il turno della versione 2.0 la quale in realtà non aggiunge grosse novità rispetto alla precedente, che si poteva già considerare una versione beta.

Nei mesi successivi sono state poi rilasciate diverse versioni minori, fino alla 2.2 (*Froyo*) che ha rappresentato un'importante pietra miliare, non tanto per le funzionalità introdotte, quanto per i primi passi verso l'ottimizzazione delle performance, che ricordiamo essere fondamentali al fine di una buona interattività. Una di queste ottimizzazioni è stata, per esempio, l'introduzione del JIT (*Just In Time compiler*) inizialmente non presente.

Le versioni 2.3.x, denominate *Gingerbread*, sono state le prime a fornire prestazioni sensibilmente migliori rispetto alle precedenti. In particolare la release 2.3.3 (ne sono state rilasciate diverse, fino all'attuale 2.3.4) è anche caratterizzata dall'introduzione di due API molto importanti, che permettono la lettura e scrittura di tag NFC (Near Field Communication) e l'accesso a servizi SIP (Session Initiation Protocol) per il Voice over IP. Ricordiamo che la possibilità di leggere e scrivere in modo sicuro informazioni di tag NFC rappresenta il primo passo verso funzionalità di micropagamento, a cui i dispositivi mobili si prestano molto. Fino alla versione 2.3.4 si è pensato ad Android come a una piattaforma per lo più software estensibile attraverso delle API, ma non dal punto di vista hardware. Dalla versione 2.3.4 è invece possibile interfacciare un dispositivo Android con altri attraverso l'interfaccia USB (Universal Serial Bus). Questo permette ai nostri dispositivi di accedere, per esempio, a informazioni acquisite da sistemi hardware esterni (per esempio Arduino) oppure di trasmettere le proprie informazioni per utilizzi personalizzati.

La velocità con cui le versioni della piattaforma vengono rilasciate da Google rappresenta sicuramente un problema. Da un lato produce una notevole frammentazione e dall'altro non permette ai produttori di dispositivi di tenere il passo con facilità. È vero che, come vedremo più avanti, la piattaforma mette a disposizione gli strumenti per tamponare il problema, ma si tratta comunque di accorgimenti che in diversi casi allungano i tempi di sviluppo e quindi i costi delle applicazioni. A questo, si è poi aggiunto il problema relativo al fatto che la piattaforma non fosse ottimizzata, sia dal punto di vista delle API, sia da quello delle caratteristiche hardware, per dispositivi tablet. Come abbiamo accennato, essi non differiscono solo per la dimensione del display, ma per le diverse modalità di interazione e quindi per le diverse tipologie di componenti visuali di cui le applicazioni hanno bisogno. Tratteremo nel dettaglio l'argomento nel Capitolo 6, ma è semplice intuire come quello che in uno smartphone deve essere visualizzato in due schermate (una di lista e una di dettaglio) possa essere invece visualizzato contemporaneamente in un tablet; la lista sulla sinistra e il dettaglio sulla destra.

La necessità di un ambiente ottimizzato per tablet ha portato al rilascio, nel febbraio del 2011 della versione 3.0 (*Honeycomb*) e successivamente 3.1 a giugno dello stesso anno. Abbiamo quindi avuto un aumento ulteriore della frammentazione, che Google ha promesso di risolvere con la prossima versione della piattaforma che prende il nome di *IceCream* e che dovrebbe quindi permettere l'utilizzo di uno stesso insieme di API per la realizzazione di applicazioni su smartphone e tablet.

Android e Java

Gli ambienti descritti in precedenza sono comunque caratterizzati da una serie di tool di sviluppo e da un linguaggio. Android non poteva essere da meno e pertanto fornisce un SDK in grado di facilitare lo sviluppo delle applicazioni. Sappiamo che la fortuna di un ambiente è legata al numero di applicazioni disponibili. È nell'interesse di Google, al fine di promuovere la piattaforma, fornire agli sviluppatori tutti gli strumenti necessari. Descriveremo in dettaglio questo ambiente nei prossimi capitoli del libro. Ciò che vogliamo sottolineare ora è invece il fatto di come il linguaggio utilizzato da Android non sia un nuovo linguaggio, che gli sviluppatori sarebbero obbligati a imparare, ma di Java, descritto dalle specifiche Sun Microsystems e che utilizziamo dal lontano 1995.

Non solo Java

Come vedremo più avanti, Google fornisce agli sviluppatori due ulteriori strumenti per lo sviluppo di applicazioni per Android: Android Scripting Environment (ASE) e Android Native Development Kit (NDK). Il primo ha l'obiettivo di semplificare lo sviluppo delle applicazioni attraverso un linguaggio di scripting di alto livello, mentre il secondo ha come obiettivo quello di sfruttare al massimo le potenzialità hardware del dispositivo, nel caso di applicazioni che richiedano un'elevata capacità di elaborazione. Fino alla versione 2.3 non era comunque possibile realizzare un'applicazione Android senza utilizzare il linguaggio Java, in quanto rappresentava l'unico modo per creare quelle che si chiamano Activity e che vedremo essere le parti fondamentali per la definizione della UI. Ora questo componente può essere invece definito utilizzando codice nativo (ovvero C++) attraverso le JNI (Java Native Interface) e quella che si chiama NativeActivity.

Nel caso di un nuovo linguaggio, la stessa Google avrebbe dovuto realizzare delle specifiche, un compilatore, un debugger, degli IDE opportuni, oltre che documentazione e librerie idonee. La scelta di Java ha però un risvolto in contrasto con quella che è la natura open di Android. I dispositivi che intendono adottare la *virtual machine* (VM) associata all'ambiente J2ME (JVM o KVM come vedremo più avanti) devono pagare una royalty, cosa in contrasto con la licenza Apache descritta in precedenza. Sarebbe come dire che Android può essere liberamente utilizzato però non funziona se non dispone della VM di Sun, la quale prevede il pagamento di una royalty. Se ci fosse poi bisogno di una KVM, tanto varrebbe, come vedremo quando confronteremo le due tecnologie, rendere Android un particolare profilo J2ME. In base a quanto detto nell'introduzione, uno degli obiettivi principali di Android è quello di creare applicazioni mobili in grado di interagire con l'utente in modo efficace. È vero che ormai quella di telefono è solo una delle funzionalità nei dispositivi di nuova generazione, ma sarebbe un problema se le altre applicazioni andassero a interrompere una telefonata in quanto necessitano, per esempio, di maggiore memoria. È indispensabile che le diverse applicazioni in esecuzione in un dispositivo Android vengano eseguite nel modo migliore dal punto di vista dell'utente e della modalità di interazione con il dispositivo. Questo aspetto, fondamentale per lo sviluppo di tutte le applicazioni mobili, non solo Android, prende il nome di *responsiveness*. Per comprendere l'importanza di questo aspetto è sufficiente dire che Apple, con l'iPhone e iTouch, pretende di poter testare l'applicazione che si intende mettere a disposizione attraverso il suo store, per evitare di pubblicare programmi che, magari per un'errata gestione della memoria, possano fornire all'utente una cattiva interazione con il dispositivo e quindi una "bad experience". Data la natura open di Android, questo aspetto viene lasciato alla coscienza dello sviluppatore, consapevole che una cattiva applicazione gli potrà portare una pessima valutazione e reputazione.

Ma come può Android eseguire byte-code Java senza utilizzare una JVM? La risposta è semplice: Android non esegue byte-code Java, per cui non ha bisogno di una JVM. Per ottimizzare al massimo l'utilizzo delle risorse dei dispositivi, Google ha adottato una propria VM che prende il nome di *Dalvik* (nome di una località in Islanda) sviluppata inizialmente da Dan Bornstein. Si tratta di una VM ottimizzata per l'esecuzione di applicazioni in dispositivi a risorse limitate, la quale esegue codice contenuto all'interno di file di estensione `.dex`, ottenuti a loro volta, in fase di building, a partire da file `.class` di byte-code Java.

Dalvik VM per le prestazioni o per la royalty a Sun?

Alcuni maliziosi sostengono che l'adozione di una virtual machine diversa dalla KVM sia dovuta a problemi di royalty a Sun. A onor del vero dobbiamo dire che, comunque, la DVM arriva a più di 10 anni dalla KVM e che era quasi una necessità approfittare di questa occasione per un minimo di ottimizzazione. La natura open di Android impedisce inoltre l'utilizzo di tecnologie con un qualche vincolo di royalty.

Le librerie standard Java sono per la quasi totalità escluse da Android; non a caso le *Abstract Window Toolkit* (AWT) e le *Swing*. La definizione dell'interfaccia grafica è, infatti, un aspetto fondamentale nell'architettura di Android, la quale, come vedremo, utilizza un approccio dichiarativo come ormai avviene nella maggior parte delle piattaforme di sviluppo. L'utilizzo della quasi totalità delle API di Java 5 permette agli sviluppatori di utilizzare le stesse librerie e classi utilizzate in precedenza in applicazioni desktop. Infatti, lo sviluppo di applicazioni per Android ricorda molto lo sviluppo di applicazioni con J2SE.

Confronto con J2ME

Visto il legame fra Android e Java, non ci si può esimere dal fare un confronto con quella che è la tecnologia che Sun ha creato per la realizzazione di applicazioni per dispositivi mobili. Ma da cosa nasce la necessità di utilizzare Java in questi dispositivi? Come sappiamo la natura di Java si riassume nella ormai famosa frase "*Write Once, run Everywhere*", che vuole esprimere il fatto che un'applicazione sviluppata in Java e compilata può essere eseguita, senza alcuna modifica, in ambienti con sistemi operativi diversi, a patto che per questi esista una specifica *Java Virtual Machine* (JVM) in grado di tradurre le istruzioni byte-code in codice nativo della specifica piattaforma. Non tutte le applicazioni sono però uguali sia dal punto di vista funzionale, ma soprattutto dal punto di vista non funzionale o dell'architettura. Un'applicazione desktop è diversa da un'applicazione web, la quale è diversa da un'applicazione in esecuzione su un cellulare. Ci si è accorti quindi della necessità di creare VM o ambienti diversi in grado di ospitare le diverse tipologie di applicazioni. Ecco che sono stati definiti gli ambienti J2SE, J2EE e J2ME ciascuno caratterizzato da una VM, librerie *runtime*, API, documentazione e soprattutto tool di sviluppo, tra cui il compilatore, il *debugger* e l'interprete. La J2SE è l'ambiente utilizzato per lo sviluppo di applicazioni desktop, mentre la J2EE contiene gli strumenti per la realizzazione di applicazioni *enterprise*, che quindi permettono l'interoperabilità con altri sistemi di vario genere (DBMS – DataBase Management System, sistemi MOM – Message Oriented Middleware, Webservice e SOA – Service Oriented Architecture). Sebbene queste siano dedicate a tipologie di applicazioni diverse tra loro, utilizzano una stessa JVM, che può essere attivata secondo due modalità di esecuzione: client e server. La modalità client favorisce l'aspetto di interazione dell'applicazione con l'utente, attraverso una veloce esecuzione e gestione degli eventi. La modalità server prevede invece un insieme di ottimizzazioni a livello di interpretazione del byte-code, al fine di gestire in modo ottimale il multithreading. Per capire quanto sia importante la gestione del multithreading in un'applicazione web, basta pensare al fatto che ciascuna richiesta HTTP da parte di un client viene gestita, appunto, da un thread. Di queste due, e dato l'insieme di librerie che si utilizzano, la J2SE è quella che vedremo assomigliare di più ad

Android. L'unica differenza, già accennata, riguarda le API per la gestione dell'interfaccia grafica, che Android gestisce in modo ottimizzato.

Infine abbiamo J2ME, che descrive l'ambiente per l'esecuzione di applicazioni in dispositivi che il più delle volte vengono associati ai cellulari. Si tratta invece di una tecnologia più complessa, che permette l'esecuzione di programmi anche in dispositivi diversi come un box per il digitale terrestre o un lettore Blu-ray.

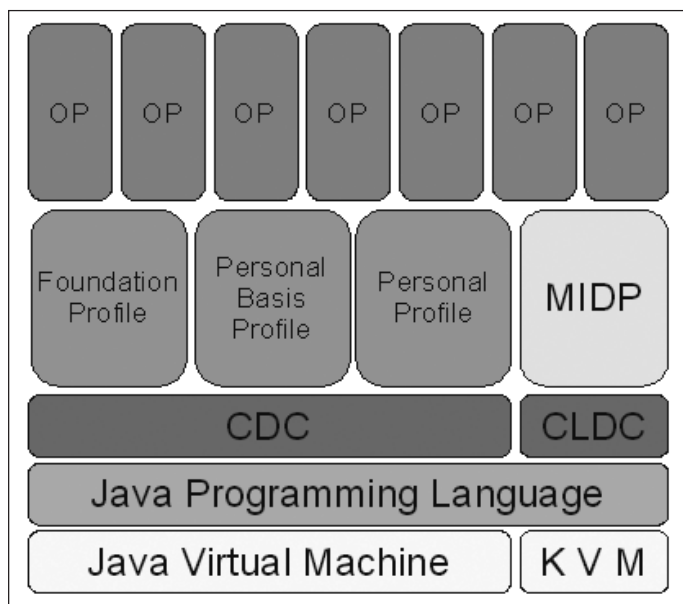


Figura 1.2 L'architettura J2ME.

Le specifiche J2ME definiscono le *configuration*, ovvero quegli ambienti per dispositivi con caratteristiche hardware e software simili. Per essere più chiari, al momento esistono solamente due tipi di *configuration*: *Connected Device Configuration* (CDC) e *Connected Limited Device Configuration* (CLDC). La CDC si riferisce a dispositivi con processore a 32 bit, una RAM di circa 2 MB e una ROM di circa 2.5 MB da dedicare all'ambiente Java. La VM di riferimento è la CDC Hotspot implementation, più nota con l'abbreviazione *Compact Virtual Machine* (CVM) di cui esistono diverse implementazioni, tra cui quelle basate su ARM, PowerPC, Linux e Solaris. La CLDC fa riferimento invece a dispositivi con memoria di almeno 192 KB, suddivisi in almeno 160 KB di memoria non volatile e 32 KB di memoria volatile. La VM di riferimento prende il nome di KVM dove la K dà un'idea delle sue dimensioni, che ne permettono l'esecuzione con una memoria di 128 KB. Notiamo subito come le dimensioni della memoria relative a una *configuration* CLDC non siano paragonabili con quelle che un'applicazione Android si aspetta di trovare in un dispositivo.

Al di sopra delle *configuration* vengono poi definiti quelli che Sun chiama *profile* e che aggiungono, alla relativa *configuration*, le API specifiche del particolare dispositivo. Per dare un'idea, la maggior parte dei cellulari ora dispone del profilo *Mobile Information Device Profile* (MIDP) nella versione 2.1, basato sulla CLDC versione 1.1. Esso aggiunge

alla CLDC le API per la gestione delle interfacce di input (nello specifico il tastierino numero o il pennino) e di output, come il display, oltre ad altre API per la realizzazione di giochi, per la gestione della persistenza e per la gestione di connessioni HTTP.

Il principale problema dell'ambiente J2ME, dal punto di vista delle prestazioni, è legato al fatto che si appoggia su una VM, la KVM, nata come sottoinsieme della JVM. Dal J2SE si è mantenuto, sia a livello di classi, sia a livello di singolo metodo di una classe, tutto ciò che poteva essere eseguito in un device e si è buttato via il resto. Non si è pensato quindi alla realizzazione di qualcosa che fosse specifico e ottimizzato per la tipologia di applicazione, come invece è avvenuto con Android.

Altro problema della J2ME si chiama *device fragmentation* ed è dovuta al fatto che anche dispositivi con lo stesso *profile* basato sulla stessa *configuration* possono avere caratteristiche molto diverse tra loro. Un dispositivo può permettere la gestione di connessioni Bluetooth e un altro no. Un dispositivo può disporre di sistemi di localizzazione e un altro no. Un dispositivo può riprodurre e acquisire determinati tipi di media e un altro no. Questa diversità viene gestita dalle specifiche attraverso quelli che si chiamano *optional package* (OT). Si tratta di librerie che possono essere presenti o meno in un dispositivo, ma che, se presenti, devono soddisfare specifiche definite nella corrispondente *Java Specification Request* (JSR). Da tutto questo si capisce come il fatto che un'applicazione venga compilata per un particolare dispositivo non assicura che un altro sia in grado di eseguirla nello stesso modo. Ecco che se uno sviluppatore intende creare un'applicazione con la tecnologia J2ME, si dovrà preoccupare di verificare l'applicazione in tutte le famiglie di dispositivi che ritiene possano essere commercialmente interessanti.

La soluzione di Sun, a questo e altri problemi della J2ME, si chiama JavaFX. Si tratta di una tecnologia nata per la realizzazione di *Rich Internet Application* (RIA), attraverso l'utilizzo di un linguaggio di scripting chiamato Java FX Script. Appoggiandosi a un linguaggio dichiarativo, JavaFX permette l'esecuzione delle applicazioni in ambienti diversi, tra cui il browser, il desktop e i dispositivi mobili. Il tutto, questa volta, senza alcuna effettiva modifica, se non nella modalità di deploy dell'applicazione. Al momento esiste un'implementazione di tale ambiente su Windows Mobile, ma presto ci si attende che venga rilasciato un ambiente anche per cellulari Symbian.

Android non ha, per il momento, il problema della frammentazione, anche se la sua caratteristica di essere completamente open non fa ben sperare da questo punto di vista. La VM poi è la *Dalvik Virtual Machine* la quale presenta caratteristiche molto importanti che, attraverso un'ossessiva cura del dettaglio, permette un utilizzo ottimale delle risorse a disposizione, come vedremo nel prossimo paragrafo.

La Dalvik Virtual Machine

Come accennato in precedenza, l'esigenza di creare applicazioni in grado di rispondere in modo immediato all'utente è fondamentale. Con un hardware a risorse limitate, non si può quindi fare altro che adottare tutti i possibili accorgimenti, a livello di architettura e software, per sfruttare al massimo le risorse disponibili.

La decisione più importante è stata quella di adottare una nuova VM, diversa da quella Sun, ottimizzata per l'esecuzione di applicazioni in ambienti a memoria ridotta, sfruttando al massimo le caratteristiche del sistema operativo ospitante. La scelta è caduta sulla *Dalvik Virtual Machine* (DVM) in grado di eseguire codice contenuto all'interno di

file di estensione `.dex` ottenuti, come accennato, a partire dal byte-code Java. Ma perché, se non si sfrutta la VM di Java non si utilizza almeno il suo byte-code, implementando solamente nuovi modi di interpretazione? La risposta sta nell'esigenza di risparmiare quanto più spazio possibile per la memorizzazione ed esecuzione delle applicazioni. Per esempio, se un'applicazione Java è descritta da codice contenuto all'interno di un archivio `.jar` di 100 KB (non compresso), la stessa potrà essere contenuta all'interno di un file di dimensione di circa 50 KB se trasformato in `.dex`. Questa diminuzione di quasi il 50% avviene nella fase di trasformazione dal byte-code Java al byte-code per la DVM, durante la quale i diversi file `dex` sono in grado di condividere informazioni che nel byte-code verrebbero ripetute più volte. Si tratta comunque della prima delle varie ottimizzazioni fatte rispetto all'esecuzione di byte-code Java.

Android vs. PC

Per fornire dati significativi, il dispositivo Android HTC Hero, di recente rilascio, ha una quantità di memoria RAM di 288 MB, una ROM di 512 MB e monta un processore Qualcomm MSM7200A (ARM) a 528MHz. Se andiamo in un qualunque sito per l'acquisto di un notebook, notiamo come un modello con caratteristiche medio basse abbia comunque almeno 2 GB di RAM e un processore Dual Core intorno ai 2 GHz.

Fare il tuning di un'applicazione Java significa, per lo più, impostare i parametri di memoria per la gestione del *Garbage Collector* (GC) ovvero di quel processo che, in base all'algoritmo implementato, permette di eliminare gli oggetti non più utilizzati, liberandone la memoria occupata. La DVM non elimina il GC, in quanto una gestione della memoria a carico del programmatore avrebbe complicato lo sviluppo delle applicazioni, ma avrebbe anche aumentato la probabilità di bug e memory leak.

Dalla versione 2.2 la DVM implementa il *Just In Time compiler* (JIT). Si tratta di un meccanismo attraverso il quale la JVM riconosce determinati pattern di codice Java, traducendoli in altrettanti frammenti di codice nativo (C e C++) per un'esecuzione più efficiente. Si trattava di una funzionalità inizialmente non implementata, in quanto la maggior parte delle API Java della piattaforma è costituita da wrapper di componenti, implementati per lo più in modo nativo. Questo significa che gran parte del codice che viene eseguito è di per sé già ottimizzato. I progettisti Google hanno comunque visto che, nella maggior parte delle applicazioni del Market, il codice Java assumeva sempre maggior importanza e quindi un'ottimizzazione della parte interpretata avrebbe comunque portato a un sensibile miglioramento delle performance. Si tratta di un'implementazione del JIT comunque ottimizzata, che entra in gioco in particolari momenti che dipendono, per esempio, dal fatto che il dispositivo sia connesso o meno all'alimentazione. Le diverse API Java che andremo a utilizzare descriveranno per lo più oggetti (wrapper) che incapsulano le funzionalità alle quali accedono attraverso la Java Native Interface (JNI) che ricordiamo essere il meccanismo che Java mette a disposizione per l'implementazione di metodi delle classi con codice nativo.

Un aspetto molto importante della DVM riguarda il meccanismo di generazione del codice che viene detto *register based* (orientato all'utilizzo di registri) a differenza di quello della JVM, detto invece *stack based* (orientato all'utilizzo di stack). Attraverso questo mec-

canismo, i progettisti della DVM si aspettano, a parità di codice Java, di ridurre del 30% il numero di operazioni da eseguire. Per capire come questo possa avvenire, supponiamo di voler valutare la seguente espressione:

```
c = a + b;
```

Se con L indichiamo un'operazione di caricamento del dato (load) e con S un'operazione di scrittura dello stesso (store), la precedente istruzione si può tradurre nelle seguenti operazioni:

```
push b;    // LS
push a;    // LS
add;       // LLS
store c;   // LS
```

Si tratta quindi del caricamento dei due operandi a e b nello stack, del calcolo della loro somma e della sua memorizzazione del risultato in cima allo stack stesso. Se volessimo eseguire la stessa operazione con un meccanismo register based otterremmo:

```
add a,b,c; // LLS
```

Si tratterebbe quindi del caricamento degli operandi a e b in zone diverse di un registro e della memorizzazione del risultato nel registro stesso. Ma quali sono i vantaggi dell'utilizzo di una tecnica piuttosto che l'altra? Sicuramente si può ottenere un minor tempo di esecuzione delle istruzioni, al prezzo di una maggiore elaborazione in fase di compilazione o trasformazione. Quelle che sono le operazioni di preparazione per l'esecuzione dell'operazione in un unico passaggio nel registro possono essere eseguite in fase di building. Capiamo quindi che, se il compilatore da byte-code Java a dex è in grado di eseguire ottimizzazioni di questo tipo, le prestazioni possono migliorare sensibilmente, a discapito di un maggior sforzo in compilazione e del maggior spazio del dex generato, che ormai, nei dispositivi attuali, non è più un problema (in ogni caso è inferiore alla corrispondente versione Java). Alleggerire il carico runtime (ovvero in fase di esecuzione) è sicuramente un aspetto positivo nei dispositivi mobili.

La DVM è quindi in grado di eseguire file dex per cui eventuali elaborazioni runtime di informazioni che prima erano disponibili all'interno di byte-code Java non saranno più possibili.

Ultima importantissima caratteristica della DVM è quella di permettere un'efficace esecuzione di più processi in contemporanea. Come vedremo, infatti, ogni applicazione sarà in esecuzione all'interno del proprio processo Linux e questo porta a vantaggi dal punto di vista delle performance e allo stesso tempo ad alcune implicazioni dal punto di vista della sicurezza.

L'architettura di Android

Come abbiamo già detto, Android è un'architettura che comprende tutto lo stack degli strumenti per la creazione di applicazioni mobili di ultima generazione, tra cui un sistema operativo, un insieme di librerie native per le funzionalità core della piattaforma,

un'implementazione della VM e un insieme di librerie Java. Si tratta di un'architettura a *layer* dove i livelli inferiori offrono servizi ai livelli superiori, con un più alto livello di astrazione. In questo e nei prossimi paragrafi faremo una panoramica di quelli che sono i principali componenti di Android, che andremo a studiare in dettaglio nei prossimi capitoli.

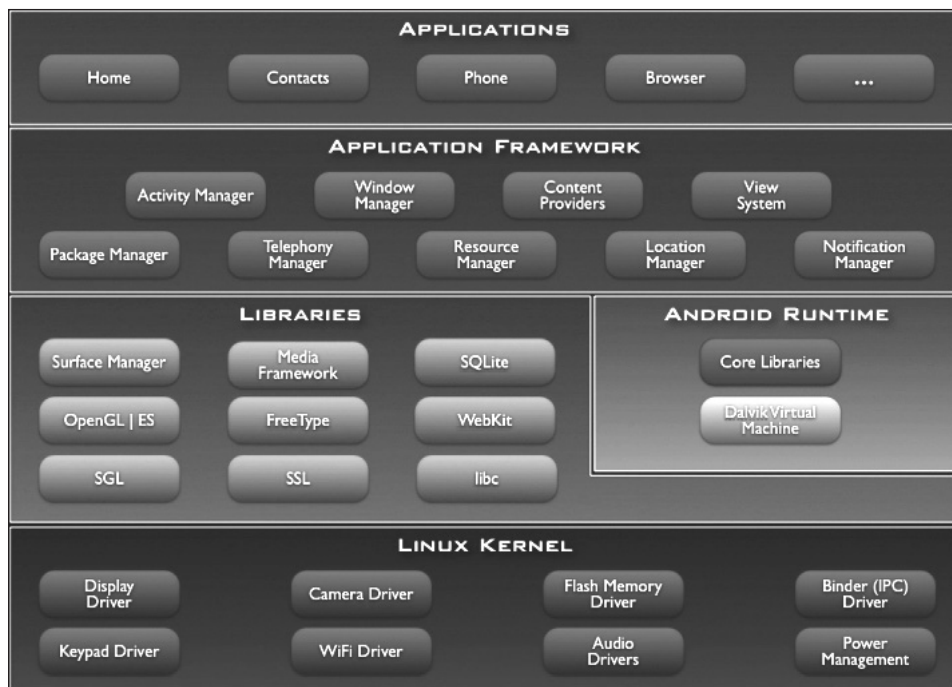


Figura 1.3 Architettura di Android (da <http://www.android.com>).

La comprensione dell'architettura descritta nella Figura 1.3, ci permetterà di comprendere al meglio l'utilizzo delle diverse funzionalità.

Il kernel di Linux

Il layer di più basso livello è rappresentato dal kernel Linux, nella versione 2.6. La necessità era, infatti, quella di disporre di un vero e proprio sistema operativo, che fornisca gli strumenti di basso livello per la virtualizzazione dell'hardware sottostante attraverso la definizione di diversi driver, il cui nome è autoesplicativo. In particolare possiamo notare la presenza di driver per la gestione delle periferiche multimediali, del display, della connessione Wi-Fi e dell'alimentazione.

Da notare anche la presenza di un driver dedicato alla gestione della comunicazione tra processi (IPC), la cui importanza è fondamentale per far comunicare i componenti in un ambiente in cui ogni applicazione viene eseguita all'interno di un proprio processo. Vedremo in dettaglio l'utilizzo di questa funzionalità quando parleremo di servizi. La scelta di utilizzare un kernel Linux deriva dalla necessità che il sistema operativo fornisca

tutte le funzionalità di sicurezza, gestione della memoria, gestione dei processi, power management e che fosse affidabile e ben collaudato. Il vendor che vorrà utilizzare Android sui propri dispositivi non dovrà quindi fare altro che installare il kernel di Linux, implementando i driver per il proprio hardware.

Librerie native

Sopra il layer costituito dal kernel 2.6 di Linux abbiamo un livello che contiene un insieme di librerie native, realizzate in C e C++, che rappresentano il core vero e proprio di Android. Si tratta di librerie che fanno riferimento a un insieme di progetti open-source che introduciamo brevemente e che approfondiremo in dettaglio per quello che riguarda le API. In questa fase sarà importante dare a ciascun elemento le proprie responsabilità. Per aiutarci, cercheremo di far sempre riferimento a concetti che già conosciamo in ambito Java standard.

Surface Manager

Il *Surface Manager* (SM) è un componente fondamentale, in quanto ha la responsabilità di gestire le *View*, ovvero ciò di cui è composta un'interfaccia grafica. Il compito del SM è quello, infatti, di coordinare le diverse finestre che le applicazioni vogliono visualizzare sullo schermo. Ogni applicazione è in esecuzione in un processo diverso e quindi disegna la propria interfaccia in tempi diversi. Il compito del SM è quello di prendere le diverse finestre e di disegnarle sul buffer, da visualizzare poi attraverso la tecnica del *double buffering*. In questo modo non si avranno finestre che si accavallano in modo scoordinato sul display. Notiamo quindi che si tratta di un componente di importanza fondamentale, specialmente in un'architettura che basa molta della sua potenza sulla capacità di creare interfacce interattive. Il *Surface Manager* avrà quindi accesso alle funzionalità del display e permetterà la visualizzazione contemporanea di grafica 2D e 3D dalle diverse applicazioni.

OpenGL ES

Un'importante caratteristica di Android, che vedremo dettagliatamente quando inizieremo a sviluppare, è la possibilità di utilizzare sia grafica 3D che 2D all'interno di una stessa applicazione. La libreria utilizzata per la grafica 3D è quella che va sotto il nome di OpenGL ES (<http://www.khronos.org/opengles/>), la quale permette di accedere alle funzionalità di un eventuale acceleratore grafico hardware. Si tratta di una versione ridotta di OpenGL specializzata per dispositivi mobili. La versione di OpenGL ES utilizzata attualmente da Android è la 2.0. Come vedremo, le API che avremo a disposizione saranno API Java. Piuttosto che reinventare la ruota, utilizzeremo API molto simili a quelle definite dalla JSR 239, che permettono di fare il binding tra Java e, appunto, le API OpenGL ES. Se andiamo a vedere le API disponibili in Android e quelle descritte dalle JSR 239 notiamo, infatti, la presenza dello stesso package `javax.microedition.khronos.opengles`. Ma cos'è l'OpenGL ES? Si tratta di un insieme di API multiplatforma che forniscono l'accesso a funzionalità 2D e 3D in dispositivi embedded. Come avvenuto per le J2ME di Java anche in relazione alla grafica su dispositivi con risorse relativamente ridotte, si è deciso di creare un sottoinsieme delle API dell'OpenGL, da cui appunto l'OpenGL ES. Un'altra analogia con le API in Java è il fatto che siano delle specifiche che i diversi produttori possono implementare adattandole alle proprie macchine o

sistemi operativi. Il tutto senza il bisogno di pagare alcuna royalty. Il target di dispositivi a cui queste API sono rivolte è caratterizzato da quantità ridotte di memoria. Si tratta, infatti, di API con un basso consumo di memoria, che può andare da 1 a 64 MB. Dalla versione 3.0 l'utilizzo di OpenGL ES per la realizzazione di mondi 3D non è l'unica opzione, in quanto sono state introdotte nuove API che prendono il nome di *Render Script*. Lo scopo di queste API è quello di semplificare e ottimizzare l'esecuzione delle operazioni di calcolo e rendering 3D.

SGL

La Scalable Graphics Library (SGL) è una libreria in C++ che, insieme alle OpenGL, costituisce il motore grafico di Android. Mentre per la grafica 3D ci si appoggia all'OpenGL, per quella 2D viene utilizzato un motore ottimizzato, chiamato, appunto, SGL. Si tratta di una libreria utilizzata principalmente dal Window Manager e dal Surface Manager all'interno del processo di rendering grafico.

Media Framework

Come abbiamo detto, la maggior parte delle applicazioni per Android è caratterizzata da un elevato utilizzo di contenuti multimediali. Per fare questo vi è la necessità di un componente in grado di gestire i diversi codec per i diversi formati di acquisizione e riproduzione audio e video, che nell'architettura di Android si chiama Media Framework. Esso è basato sulla libreria open-source OpenCore di PacketVideo, uno dei membri fondatori della OHA ovvero di quel gruppo di società che intendono investire nella definizione di dispositivi mobili sempre più efficienti e sempre meno costosi di cui Android è un'espressione.

La versione di OpenCore utilizzata dal SDK 3.1 è la 2.5, la quale, oltre a disporre di una più efficiente gestione dei buffer e di un maggior numero di codec, contiene il supporto agli encoder OpenMax.

OpenMax

Si tratta di un insieme di API, liberamente utilizzabili, che permettono un'astrazione delle operazioni che un dispositivo mobile è in grado di eseguire su un determinato stream di dati. Si tratta di un passo verso la standardizzazione nella gestione dei codec, al fine di diminuire il problema della segmentazione dei dispositivi.

I codec gestiti dal Media Framework permettono la gestione dei formati più importanti, tra cui MPEG4, H.264, MP3, AAC, AMR oltre a quelli per la gestione delle immagini, come JPG e PNG. Ovviamente nuovi formati potranno essere supportati nei dispositivi che, di volta in volta, vengono immessi sul mercato.

FreeType

La gestione dei font è un altro aspetto molto importante nella definizione di un'interfaccia. A tale proposito, per Android si è scelto di utilizzare il motore di rendering dei font

FreeType (<http://freetype.sourceforge.net/>). Si è deciso di utilizzare questo motore in quanto di piccole dimensioni, molto efficiente, altamente personalizzabile e soprattutto portabile. Attraverso FreeType, le applicazioni di Android saranno in grado di visualizzare immagini di alta qualità. Il suo punto di forza è quello di fornire un insieme di API semplici per ciascun tipo di font in modo indipendente dal formato del corrispondente file. Di default i principali tipi di formati di font utilizzati sono TrueType, Type1, X11 PCF e OpenType, insieme ad altri più recenti.

SQLite

Chi sviluppa con le MIDP conoscerà molto bene il *Record Management System* (RMS) ovvero quel sistema che permette di memorizzare in modo persistente sul dispositivo una certa quantità di informazioni. Nelle MIDP 1.0 il problema era legato principalmente al poco spazio a disposizione, tipicamente sui 20 KB. Nelle MIDP 2.0, grazie anche alle caratteristiche dei dispositivi sul mercato, la memoria dedicata al RMS è vincolata solo alla dimensione della memory card inserita nel dispositivo (ormai anche 4 o 8 GB). Le RMS sono però abbastanza complesse. Ricordiamo come per filtrare un certo insieme di informazioni sia necessario creare implementazioni di varie interfacce descritte dalla API MIDP 2.0. A questo si aggiunge la difficoltà introdotta dal fatto che ciascun record viene gestito come array di byte; ciò costringe alla continua serializzazione e deserializzazione delle informazioni. Forse alla luce di queste difficoltà, con Android si è deciso di utilizzare qualcosa di comunque efficiente e compatto, ma che avesse per quanto possibile le caratteristiche di un DBMS relazionale. Da qui la decisione di utilizzare SQLite (<http://www.sqlite.org>) ovvero una libreria in-process che implementa un DBMS relazionale caratterizzato dal fatto di essere molto compatto, diretto, di non richiedere alcuna configurazione e, soprattutto, transazionale. SQLite è compatto, in quanto realizzato completamente in C in modo da utilizzare solo poche funzioni ANSI per la gestione della memoria. È diretto, in quanto non utilizza alcun processo separato per operare, ma “vive” nello stesso processo dell’applicazione che lo usa, da cui il termine in-process. Viene utilizzato spesso in sistemi embedded, in quanto non necessita di alcuna procedura di installazione. Nonostante la compattezza e, come vedremo quando ci occuperemo delle sue API, la semplicità d’uso, permette di gestire gli accessi ai dati in modo transazionale. Da sottolineare come SQLite non sia un prodotto di Android, ma esista autonomamente con diversi tool a supporto. Gran parte delle librerie e degli strumenti che si utilizzano con SQLite 3 può essere utilizzata anche nel con SQLite in Android.

WebKit

Nell’era del Web 2.0 non poteva di certo mancare un browser integrato nella piattaforma. A tale proposito la scelta è andata sul framework WebKit (<http://webkit.org>) utilizzato anche dai browser Safari e Chrome. Si tratta di un browser-engine open-source basato sulle tecnologie HTML, CSS, JavaScript e DOM. Un aspetto da sottolineare è che WebKit non è un browser, ma un browser-engine e che quindi andrà integrato in diversi tipi di applicazioni. Da notare come questo motore sia molto importante anche per il fatto che la modalità di presentazione e di interazione con un browser in un PC sono sicuramente diverse da quelle che si possono avere su un dispositivo generalmente piccolo con tastiera limitata come un cellulare o altro dispositivo mobile.

SSL

Si tratta dell'ormai famosa libreria per la gestione dei Secure Socket Layer. Anche gli aspetti legati alla sicurezza non potevano di certo essere trascurati da Android.

Libc

Si tratta di un'implementazione della libreria standard C libc ottimizzata per i dispositivi basati su Linux embedded, come Android.

Le core library

Come sappiamo, per eseguire un'applicazione in Java non serve solamente l'applicazione in sé, ma anche tutte le classi relative all'ambiente in cui viene eseguita. Per quello che riguarda la J2SE, stiamo parlando delle classi contenute nel file `rt.jar` ovvero quelle relative ai package `java` e `javax`. Per le applicazioni Android vale lo stesso discorso, con la differenza che, in fase di compilazione, avremo bisogno del `jar` (di nome `android.jar`) per la creazione del byte-code Java, mentre in esecuzione il device metterà a disposizione la versione dex del runtime, che costituisce appunto la core library. Nel dispositivo non c'è, infatti, codice Java, in quanto non potrebbe essere interpretato da una JVM, ma solamente codice dex eseguito dalla DVM. Abbiamo già visto poi come queste librerie siano per la maggior parte la descrizione di componenti wrapper per l'accesso a funzionalità implementate in modo nativo.

Application framework

Tutte le librerie viste finora vengono utilizzate da un insieme di componenti di più alto livello, che costituiscono l'Application Framework (AF). Si tratta di un insieme di API e componenti per l'esecuzione di funzionalità ben precise e di fondamentale importanza in ogni applicazione Android. Anche in questo caso ne diamo una rapida introduzione, prima di vederli in dettaglio nei prossimi capitoli. Ultima importante nota: tutte le applicazioni per Android utilizzano lo stesso AF e come tali possono essere estese, modificate o sostituite. Da qui il motto che possiamo trovare sul sito di Android ovvero: "All applications are equals".

Activity Manager

Quando inizieremo lo sviluppo delle applicazioni per Android, vedremo come assuma fondamentale importanza il concetto di *Activity*. Si tratta di un qualcosa che possiamo associare inizialmente a una schermata, che quindi permette non solo la visualizzazione o la raccolta di informazioni, ma, in modo più generico, è lo strumento fondamentale attraverso il quale l'utente interagisce con l'applicazione. Successivamente vedremo come sia fondamentale capire in dettaglio quello che è il ciclo di vita, gestito dall'*Activity Manager*, di ciascuna activity. Responsabilità di questo componente sarà quindi l'organizzazione delle varie schermate di un'applicazione in uno stack, a seconda dell'ordine di visualizzazione delle stesse sullo schermo dei diversi dispositivi. Comprendere quindi a fondo il funzionamento di questo componente è un aspetto fondamentale nella realizzazione delle applicazioni. Per questo motivo dedicheremo all'argomento il Capitolo 4.

Package Manager

Un aspetto non trascurabile di un sistema come Android è la gestione del processo di installazione delle applicazioni nei dispositivi. Come vedremo, ogni applicazione dovrà fornire al dispositivo che le andrà a eseguire, un determinato insieme di informazioni che descriveremo attraverso un opportuno file XML di configurazione che prende il nome di *AndroidManifest*. Si tratta di informazioni di vario genere, quali quelle relative agli aspetti grafici (il layout) dell'applicazione, alle diverse *Activity* o ad aspetti di sicurezza. La responsabilità del *Package Manager* sarà quindi quella di gestire il ciclo di vita delle applicazioni nei dispositivi, analogamente a quanto avviene in ambiente J2ME da parte del *Java Application Manager* (JAM).

Window Manager

Un componente molto importante è quindi il *Window Manager*, il quale permette appunto di gestire le finestre delle diverse applicazioni, gestite da processi diversi, sul display del dispositivo. Esso si può considerare come un'astrazione, con API Java, dei servizi nativi del *Surface Manager*, descritto in precedenza.

Telephony Manager

Se ci pensiamo, nella piattaforma MIDP 2.0 non esistono API che facciano un riferimento esplicito al fatto che le applicazioni vengono eseguite per lo più all'interno di telefoni cellulari. Per Android non è così, in quanto è disponibile il *Telephony Manager*, che permette una maggiore interazione con le funzionalità caratteristiche di un telefono, come la semplice possibilità di iniziare una chiamata o di verificare lo stato della chiamata.

Content Provider

Il *Content Provider* (CP) è un componente fondamentale nella realizzazione delle applicazioni per Android, in quanto ha la responsabilità di gestire la condivisione di informazioni tra i vari processi. Il funzionamento è simile a quello di un repository condiviso, con cui le diverse applicazioni possono interagire inserendo o leggendo informazioni. Nel Capitolo 8 vedremo come utilizzare un *Content Provider* e come realizzarne uno.

Resource Manager

Come vedremo in dettaglio, un'applicazione è composta, oltre che da codice, anche da un insieme di file, per esempio immagini, file di configurazione o di proprietà per l'internazionalizzazione (I18N), file di definizione del layout e così via. La responsabilità di gestire questo tipo di informazioni è affidata al *Resource Manager*, che mette a disposizione una serie di API di semplice utilizzo. Si tratta di un componente che ha il compito di ottimizzare le risorse. Come avviene per il codice da eseguire, anche per le risorse esiste un processo di trasformazione in contenuti binari, ottimizzati per il loro utilizzo all'interno di un dispositivo. Se la risorsa è, per esempio, un documento XML, questo non verrà installato nell'applicazione in modo testuale, ma in un formato ottimizzato rispetto alla principale operazione possibile, ovvero il parsing. Un concetto analogo vale per le altre tipologie di risorse che Android è in grado di gestire, come immagini, animazioni, menu e altre che vedremo più avanti. Altro aspetto fondamentale riguarda la possibilità

di accedere a queste risorse attraverso costanti generate in modo automatico in fase di building. Il Capitolo 3 è dedicato proprio alla gestione delle risorse.

View System

Come vedremo più avanti, l'interfaccia grafica di un'applicazione Android è composta da quelle che saranno specializzazioni della classe *View* ciascuna caratterizzata da una particolare forma e da un diverso modo di interagire con essa attraverso un'accurata gestione degli eventi associati. La gestione del rendering dei componenti, nonché della gestione degli eventi associati, è di responsabilità di un componente che si chiama *View System* (VS). Potremmo quindi fare un'analogia tra il VS e i meccanismi di gestione delle GUI della J2SE, quali AWT o Swing.

Location Manager

Tra le diverse demo disponibili delle applicazioni Android, quelle relative alla gestione delle mappe sono sicuramente tra le più interessanti. Le applicazioni che gestiscono, tra le informazioni disponibili, quelle relative alla localizzazione si chiamano *Location Based Application* (LBA) e possono essere realizzate utilizzando le API messe a disposizione dal Location Manager. Per quei dispositivi che lo permettono (che saranno progressivamente la maggioranza), potremo quindi accedere a funzionalità legate alla location, tra cui operazioni di georeferenziazione, come vedremo nel Capitolo 11.

Notification Manager

Un altro servizio molto utile è quello fornito dal *Notification Manager*, che potrebbe sembrare la versione Android del Push Registry delle MIDP 2.0. Mentre il Push Registry permette l'attivazione di un'applicazione a seguito di un particolare evento, come l'arrivo di un SMS, il Notification Manager mette a disposizione un insieme di strumenti che l'applicazione può utilizzare per inviare una particolare notifica al dispositivo, il quale la dovrà notificare all'utente con i meccanismi che conosce. L'applicazione può quindi notificare un particolare evento al dispositivo, il quale potrebbe, per esempio, emettere una vibrazione, far lampeggiare i led, visualizzare un'icona (come per gli SMS, per intenderci) e altro ancora.

Conclusioni

In questo primo capitolo abbiamo illustrato il contesto nel quale Android nasce e intende affermarsi come ambiente per la realizzazione di applicazioni mobili. Abbiamo visto in dettaglio quelle che sono state le principali motivazioni che hanno portato i progettisti di Google alla scelta di Java come linguaggio di programmazione, ma all'uso della DVM come VM per l'esecuzione. Abbiamo poi elencato gli elementi principali dell'architettura di Android e che saranno argomento dei prossimi capitoli, nei quali studieremo in dettaglio le relative API.

Nel prossimo capitolo configureremo l'ambiente di sviluppo e inizieremo a creare le nostre prime applicazioni per Android.