Capitolo 1

Sviluppare Mobile

In questo capitolo

- **Il problema della frammentazione**
- **Gestire schermi di dimensioni diverse**
- **Conclusioni**

Il precedente libro, *Android. Guida per lo sviluppatore* edito da Apogeo a febbraio 2010 ha avuto come obiettivo quello di descrivere le principali caratteristiche della piattaforma Android ponendo particolare attenzione allo studio delle API e degli strumenti che gli sviluppatori di Google ci hanno messo a disposizione per la realizzazione delle nostre applicazioni. Siamo passati da un anno, il 2009, caratterizzato da un repentino susseguirsi di versioni di Android a un anno di calma apparente che ha avuto nel rilascio della versione 2.2 della piattaforma, in codice Froyo, il suo momento di maggior interesse. A oggi è stata appena rilasciata la versione 2.3, denominata Gingerbread, la quale ha avuto come obiettivo principale l'ottimizzazione delle risorse per favorire al massimo quella che va sotto il nome di *user experience*. Dal punto di vista delle funzionalità, gli aspetti di maggior interesse di questa piattaforma riguardano l'introduzione delle API per la gestione del protocollo SIP per le chiamate voice over IP oltre che quelle per la gestione dei Near Field Communication (NFC). Da pochi giorni è avvenuto poi il rilascio di una preview della versione 3.0 che ha come principale caratteristica quella di essere studiata per i dispositivi tablet ovvero dotati di schermi più grandi con lo scopo di essere un utile strumento di lettura e di lavoro. È quindi semplice intuire come i dispositivi in grado di ospitare questo ormai famoso ambiente non saranno solamente cellulari o smartphone ma anche device con caratteristiche tecniche diverse perché studiati per fornire funzionalità diverse. In questo primo capitolo vedremo i principali aspetti di cui uno sviluppatore mobile dovrà tenere conto

nella realizzazione di un'applicazione Android al fine di coprire, con il minimo sforzo, la gamma più ampia di dispositivi. Si tratta infatti di un aspetto fondamentale in quanto maggiore è il numero di dispositivi in grado di eseguire un'applicazione, maggiore sarà il numero di copie installate e quindi vendute.

Il problema della frammentazione

Statisticamente il motivo principale del fallimento di un'applicazione mobile è dato dal tentativo di realizzare una unica versione per una gamma molto ampia di dispositivi. La tecnologia J2ME ci ha insegnato molte cose al riguardo a causa del noto problema di frammentazione. Il fatto che un dispositivo J2ME sia compatibile con la versione 2.0 del profilo MIDP, garantisce solamente la possibilità di poter eseguire un insieme di API che rappresentano ormai un piccolo sottoinsieme della totalità di API disponibili che, come sappiamo, sono descritte da quelli che, in un contesto J2ME, si chiamano Optional Package. Non ha infatti più senso realizzare una `MIDlet` (applicazione MIDP) che non sfrutti appieno le caratteristiche della maggior parte dei dispositivi che al giorno d'oggi sono dotati di supporto Bluetooth, giroscopio, bussola, GPS e altro ancora. Il problema è legato più che altro al fatto che non tutte queste funzionalità sono disponibili in tutti i device MIDP. Android ha fatto tesoro di questo insegnamento e ha preso due importanti decisioni.

1. *Tutte* le API relative a un particolare API Level sono disponibili in *tutti* i dispositivi che supportano tale versione.
2. Un device che accede al Market vedrà solamente quelle applicazioni che lo stesso è in grado di eseguire senza problemi.

Il primo punto potrebbe sembrare la dichiarazione che nel mondo Android non vi sia frammentazione e che quindi tutti i dispositivi in grado di supportare una particolare versione siano dotati delle stesse caratteristiche hardware. In realtà un API Level non è strettamente legato a caratteristiche hardware del dispositivo ma, come dice il nome stesso, alle sue API e quindi a un aspetto software. Ricordiamo che un API Level è un valore numerico che identifica in modo univoco una particolare versione della piattaforma di Android. In particolare ciascun API Level definisce insiemi di:

- librerie e classi;
- elementi XML che permettono la definizione del file `AndroidManifest.xml` che sappiamo descrivere ciascuna applicazione al dispositivo;
- elementi XML che permettono la definizione e l'utilizzo di risorse;
- Intent predefiniti per la comunicazione tra componenti diversi della piattaforma;
- `permission` che l'applicazione dovrà dichiarare per poter utilizzare quelle funzionalità che le richiedono.

Il punto 1 afferma quindi che se due dispositivi Android sono compatibili con un particolare API Level allora disporranno di tutte le librerie, risorse, intent e permessi che lo stesso definisce: nessuna è opzionale per la piattaforma. Abbiamo però detto che la frammentazione esiste anche in Android ovvero che un dispositivo può avere determinati elementi hardware che un altro dispositivo non ha sebbene faccia riferimento allo stesso API Level. Per risolvere questo problema Android mette a disposizione dello sviluppatore

una serie di strumenti per impedire che un dispositivo si trovi a eseguire delle applicazioni che prevedono l'accesso a funzionalità hardware di cui lo stesso dispositivo non è dotato. Si tratta degli stessi meccanismi che l'Android Market utilizza per fare in modo che un device possa vedere solamente quelle applicazioni che lo stesso è in grado di eseguire.
Il meccanismo alla base di tutto questo è molto semplice oltre che simile a quello utilizzato dai browser per la determinazione dei tipi di dato che lo stesso è in grado di visualizzare. Una possibilità poteva essere quella di specificare per ciascuna applicazione l'insieme dei device supportati. Un approccio di questo tipo sarebbe molto difficile da gestire specialmente se pensiamo al numero di dispositivi Android esistenti e soprattutto alla frequenza con cui i nuovi modelli vengono messi continuamente sul mercato. Piuttosto che basarsi sui tipi di dispositivi a cui una particolare applicazione è dedicata, è preferibile concentrarsi sull'insieme di funzionalità (*feature*) che la stessa applicazione richiede per l'esecuzione. In sintesi ogni applicazione dovrà specificare all'interno del proprio `AndroidManifest.xml`, l'insieme delle specifiche funzionalità di cui ha bisogno attraverso la definizione di elementi `<uses-features/>`. Si tratta di elementi caratterizzati da un feature `id` e da una informazione di obbligatorietà.

Listato 1.1 Esempio di definizione di feature supportate nell'AndroidManifest.xml

```
<uses-feature android:name="android.hardware.bluetooth" android:required="false" />
<uses-feature android:name="android.hardware.camera.autofocus"
android:required="false" />
<uses-feature android:name="android.hardware.camera" android:required="true" />
```

Alcune feature saranno infatti obbligatorie per una data applicazione mentre altre saranno opzionali ed eventualmente disabilitate a runtime. Per esempio, un'applicazione che permette l'acquisizione di immagini necessiterà della presenza di una videocamera mentre la possibilità di condividere le foto acquisite attraverso una connessione Bluetooth potrà essere una funzionalità aggiuntiva che l'applicazione abilita o meno a seconda che la corrispondente feature sia supportata dal dispositivo oppure no. Questo, come vedremo, presuppone anche l'esistenza di un insieme di API che permettano di interrogare il dispositivo sulle sue effettive capacità. Dal precedente esempio si nota come ciascuna feature sia individuata da un nome i cui valori sono descritti dalle feature `id` della Tabella 1.1.

Tabella 1.1 Feature id relative alle API Level 8 per l'hardware

Feature id	Descrizione
`android.hardware.bluetooth`	L'applicazione utilizza le funzionalità relative a una connessione Bluetooth.
`android.hardware.camera`	L'applicazione utilizza un videocamera. Nel caso ve ne fossero più di una, il dispositivo utilizzerà quella nella parte posteriore del device.
`android.hardware.camera.autofocus`	Nel caso in cui sia obbligatoria la presenza di videocamera, è possibile aggiungere questa feature relativa alla disponibilità di autofocus.

(continua)

Tabella 1.1 *(segue)*

`android.hardware.camera.flash`	Nel caso in cui sia obbligatoria la presenza di una videocamera, è possibile aggiungere questa feature relativa alla disponibilità del flash.
`android.hardware.location`	Il dispositivo è in grado di utilizzare sistemi diversi per la determinazione della location.
`android.hardware.location.network`	Se il dispositivo è obbligato a gestire la location, indica la possibilità di acquisire la location attraverso le funzionalità di Rete.
`android.hardware.location.gps`	Se il dispositivo è obbligato a gestire la location, indica la possibilità di acquisire la location attraverso le funzionalità GPS.
`android.hardware.sensor.accelerometer`	Disponibilità di un accelerometro.
`android.hardware.sensor.compass`	Disponibilità di una bussola (compass).
`android.hardware.sensor.light`	Disponibilità dei sensori luminosi.
`android.hardware.sensor.proximity`	Disponibilità dei sensori di prossimità.
`android.hardware.microphone`	Disponibilità di un microfono.
`android.hardware.telephony`	Disponibilità delle funzionalità legate all'utilizzo del telefono.
`android.hardware.telephony.cdma`	Disponibilità delle funzionalità legate all'utilizzo del telefono attraverso CDMA.
`android.hardware.telephony.gsm`	Disponibilità delle funzionalità legate all'utilizzo del telefono attraverso GSM.
`android.hardware.touchscreen`	L'applicazione utilizza le funzionalità legate alle funzioni di touch.
`android.hardware.touchscreen.multitouch`	L'applicazione utilizza le funzionalità multitouch.
`android.hardware.touchscreen.multitouch.distinct`	L'applicazione utilizza funzionalità multitouch avanzate come per esempio la possibilità di gestire più punti di tocco in modo indipendente.
`android.hardware.wifi`	L'applicazione utilizza le funzionalità relative al protocollo 802.11 (WiFi).
`android.software.live_wallpaper`	L'applicazione intende utilizzare o fornire dei live wallpaper. A differenza dei valori precedenti, questo definisce una feature software e non hardware che l'applicazione intende utilizzare.

A queste Gingerbread ha poi aggiunto le seguenti altre feature disponibili quindi dall'API Level 9 identificativo di questa versione.

Tabella 1.2 Feature id relative alle API Level 9

Feature id	Descrizione
`android.hardware.audio.low_latency`	Il dispositivo utilizza un sistema di acquisizione o riproduzione audio a bassa latenza.

`android.hardware.camera.front`	Una novità della versione 2.3 riguarda la gestione di più videocamere. Questa feature indica la presenza di una videocamera frontale.
`android.hardware.nfc`	Questa feature descrive l'utilizzo di strumenti per la lettura di tag `NFC` da parte del dispositivo.
`android.hardware.sensor.barometer`	Questa feature descrive l'utilizzo delle funzionalità del sensore di pressione.
`android.hardware.sensor.gyroscope`	Questa feature descrive l'utilizzo delle informazioni provenienti dal giroscopio del dispositivo.
`android.hardware.sip`	Indica che l'applicazione utilizza le API per la gestione dello stack SIP.
`android.hardware.sip.voip`	Indica che l'applicazione utilizza le API per la gestione dello stack SIP per servizi di voice over IP.
`android.hardware.touchscreen.multitouch.jazzhand`	Indica che l'applicazione utilizza funzionalità avanzate di multitouch che prevedono l'utilizzo di cinque o più tocchi.

Si tratta di valori definiti da ciascun API Level che ovviamente dovranno essere condivisi tra coloro che dichiarano delle capacità e coloro che invece ne verificano la presenza.

Attributo glEsVersion

Oltre agli attributi `android:name e android:required`, l'elemento `<uses-features/>` definisce anche l'attributo `android:glEsVersion` il quale permette di specificare la versione minima di OpenGL ES necessaria all'esecuzione dell'applicazione. Mentre il valore di default per `android:required` è `true`, quello di `glEsVersione` è quello relativo alla versione 1.0.

Da quanto detto capiamo che i dispositivi che accedono al Market dovranno informare lo stesso di quelle che sono le feature supportate. Ecco che il Market non farà altro che incrociare le feature richieste dall'applicazione con quelle disponibili nel device mostrando solo le applicazioni compatibili. Un dispositivo che accede al Market non potrà scaricare un'applicazione che lo stesso non è in grado di eseguire. Un aspetto molto importante riguarda il fatto che si tratta di una gestione che non implica alcuna conoscenza da parte dell'utente il quale sarà in grado di eseguire solo applicazioni compatibili con il proprio dispositivo.
A questo punto il lettore potrà correttamente sollevare una eccezione relativa al fatto che la logica descritta viene comunque implementata a livello di Market e che quindi non esiste un meccanismo analogo implicito nel sistema. In effetti quando l'applicazione Android Market viene eseguita all'interno di un dispositivo, interroga il sistema richiedendo l'insieme delle feature che lo stesso è in grado di supportare. Per fare questo utilizza i servizi offerti dalla classe `PackageManager` del package `android.content.pm` tra cui quello descritto dal metodo seguente:

```
public abstract FeatureInfo[] getSystemAvailableFeatures ()
```

il quale ritorna un array di oggetti di tipo `FeatureInfo` che incapsulano le informazioni relative a ciascuna feature viste in precedenza e quindi descritte in termini di un nome e un flag che ne indica l'obbligatorietà o meno. Sono esattamente le informazioni che un'applicazione descrive attraverso uno o più elementi di tipo `<uses-features/>`.
Come esempio di quanto detto realizziamo una semplice applicazione che permette di accedere alle informazioni relative alle feature di cui un dispositivo è dotato. Creiamo quindi il progetto `TestFeatureApp` che il lettore potrà trovare nel codice allegato alla scheda del libro sul sito Apogeo, disponibile all'indirizzo `http//www.apogeonline.com/libri/9788850330102/scheda`. Si tratta di un'applicazione molto semplice che ha un'unica `Activity` la quale accede alle informazioni del dispositivo visualizzandole all'interno di una lista. A parte l'implementazione dell'`Adapter` che utilizza il pattern Holder che vedremo quando studieremo le varie modalità di utilizzo di una `ListView`, possiamo notare le seguenti istruzioni.

Listato 1.2 Interrogazione sulle feature di un dispositivo

```
// Otteniamo il riferimento al PackageManager
PackageManager packageManager = getPackageManager();
// Accediamo all'insieme di FeatureInfo
FeatureInfo[] features = packageManager.getSystemAvailableFeatures();
```

Innanzitutto otteniamo il riferimento al `PackageManager` attraverso il metodo `getPackageManager()` che la nostra `Activity` eredita per il fatto di essere una specializzazione della classe `Context`. Successivamente otteniamo l'elenco delle informazioni incapsulate all'interno di un array di oggetti di tipo `FeatureInfo`. Di seguito non facciamo altro che visualizzare le informazioni in una lista. Per testarne il funzionamento non ci resta che eseguire l'applicazione all'interno di un qualche dispositivo. Nelle seguenti figure sono visualizzati, per esempio, i risultati che si ottengono rispettivamente in un Samsung Galaxy con Froyo (e quindi la 2.2) e in un HTC MyTouch 3G con la versione 2.1 update 1 della piattaforma.

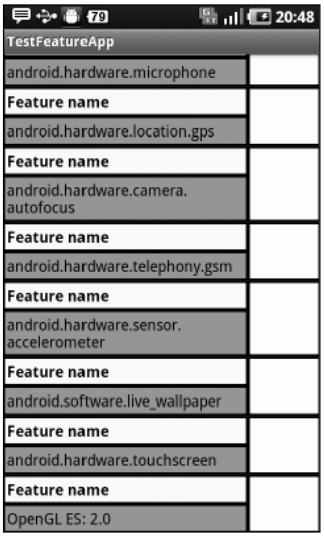

Figura 1.1 Samsung Galaxy con Froyo.

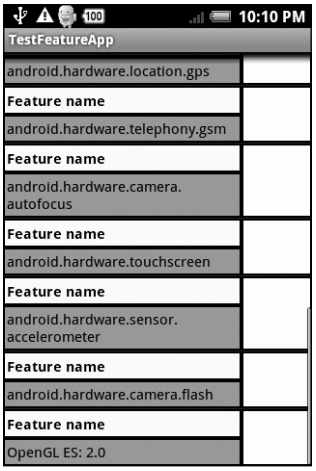

Figura 1.2 HTC MyTouch 3G con Android 2.1 update 1.

Dalle immagini possiamo notare come il secondo dispositivo sia in grado di gestire anche applicazioni che fanno uso del flash a differenza del Galaxy che permette, invece, di gestire un sensore di prossimità cosa che il MyTouch non fa. L'aspetto comunque importante è che nessuno dei due dispositivi sarà in grado di scaricare dall'Android Market un'applicazione che supporta funzionalità che non è in grado di gestire.
Il lettore potrà verificare come l'esecuzione della precedente applicazione con l'emulatore non porti alla visualizzazione di un elenco di feature molto nutrito. Si consiglia quindi di provarla su un dispositivo reale.

Gestione delle feature a runtime

Nel precedente esempio abbiamo visto come sia possibile determinare l'insieme delle funzionalità che un particolare dispositivo è in grado di supportare e di come queste informazioni possano essere utilizzate per impedire l'installazione di programmi non funzionanti in un particolare device. Abbiamo altresì ricordato come una specifica funzionalità hardware possa essere in alcuni casi opzionale e di come sia responsabilità della stessa applicazione abilitarla o meno. Per ottenere questo obiettivo l'applicazione non dovrà fare altro che verificare se una particolare funzionalità è supportata dal dispositivo. Come ottenere l'insieme delle funzionalità è stato descritto in precedenza, ora vediamo di risolvere il problema inverso ovvero: data la funzionalità verificare se è supportata. Anche in questo caso si tratta di un problema facilmente risolvibile grazie al metodo:

```
public abstract boolean hasSystemFeature (String name)
```

della classe `PackageManager`. Il parametro da passare a questo metodo sarà il nome della feature di cui si intende verificare la compatibilità. Per semplificare il tutto, ciascuna API Level definisce nella classe `PackageManager` un insieme di costanti statiche, ciascuna delle quali corrisponde a un possibile feature `id`. Ecco che nel caso in cui volessimo verificare se il dispositivo supporta le connessioni Bluetooth sarà sufficiente eseguire le seguenti righe di codice che, per fare ordine,abbiamo incapsulato all'interno di un metodo.

Listato 1.3 Verifica della disponibilità di connessioni Buetooth

```
private boolean checkForBluetooth(){
    // Otteniamo il riferimento al PackageManager
    PackageManager packageManager = getPackageManager();
    // Verifichiamo se abilitato
    boolean bluetoothSupported = packageManager.hasSystemFeature(PackageManager.
    FEATURE_BLUETOOTH);
    // Ritorniamo il valore ottenuto
    return bluetoothSupported;
}
```

Abbiamo quindi evidenziato l'utilizzo del metodo `hasSystemFeature()` insieme alla costante `FEATURE_BLUETOOTH` della classe `PackageManager`. Ovviamente la stessa operazione è possibile per ciascuna delle feature disponibili per il particolare API Level.

Compatibilità e Api Level

Concludiamo quindi questo primo paragrafo domandandoci quale sia l'incidenza dell'`Api Level` supportate da un dispositivo nella selezione delle applicazioni che lo stesso potrà eseguire. In sintesi, se un dispositivo supporta il livello 8 delle API che corrisponde a Froyo, ovvero alla versione 2.2 dell'SDK, quali sono le applicazioni che lo stesso potrà eseguire? Come regola generale ciascun dispositivo associato a un livello N di API è in grado di eseguire, fatto salvo le considerazioni relative alle feature dei paragrafi precedenti, tutte le applicazioni scritte per le versioni M <= N. Questo perché esiste la volontà di mantenere una compatibilità all'indietro e quindi fare in modo che se un'applicazione funziona con la 1.6 funzioni anche con la 2.2 o la 2.3 o successive. Ovviamente è possibile che qualche API venga modificata o migliorata nel tempo ma questo ha portato, fino a ora, a marcare alcune classi o singoli metodi come `deprecated` senza quindi eliminarli. Gli sviluppatori sanno comunque che, per le nuove applicazioni, è bene non utilizzare classi e metodi obsoleti anche perché se sono diventati tali significa che esistono soluzioni migliori che quelle legate al loro uso.
In ogni caso Android permette agli sviluppatori di descrivere l'applicazione in termini di::

- livello minimo di API su cui l'applicazione riesce a funzionare;
- livello di API per il quale l'applicazione è stata progettata e realizzata;
- livello massimo di API su cui l'applicazione riesce a funzionare.

Si tratta di informazioni che vengono specificate nell'`AndroidManifest.xml` dell'applicazione attraverso opportuni attributi dell'elemento `<uses-sdk/>` che inseriamo all'interno dell'elemento `<manifest/>`. Il livello minimo supportato da un'applicazione viene indicato attraverso l'attributo

```
android:minSdkVersion
```

Si tratta di una informazione molto importante in quanto descrive appunto il minimo livello di API che un dispositivo deve avere per poter eseguire l'applicazione. Se non specificato, il suo valore di default è 1 e indica che l'applicazione può essere eseguita da

tutte le versioni di Android. Se però essa utilizza delle API che non sono presenti nel dispositivo, il risultato sarà il crash dell'applicazione in quanto tenta di utilizzare classi o invocare metodi inesistenti. È bene quindi fare attenzione al valore specificato attraverso questo attributo anche alla luce degli eventuali aggiornamenti dell'applicazione. Le nuove versioni di un programma dovranno avere un valore di `minSdkVersion` (versione minima), uguale o minore a quella che la stessa applicazione aveva nelle versioni precedenti. Questo per impedire che chi ha acquistato una versione dell'applicazione non riesca più a eseguirla a seguito di un aggiornamento. Si tratta poi di una informazione che viene utilizzata come filtro analogamente a quello che avviene con le feature descritte in precedenza. Programmi come l'`AndroidMarket` non metteranno a disposizione di device che supportano un particolare livello di API, applicazioni che necessitano di un livello minimo di API superiore.
Oltre a quella minima è possibile specificare anche la versione presa come riferimento nella realizzazione dell'applicazione attraverso l'attributo:

```
android:targetSdkVersion
```

Si tratta di quella informazione che definisce i possibili elementi che possiamo utilizzare nella descrizione dello stesso documento `AndroidManifest.xml`. Permette di indicare che l'applicazione è stata testata con dispositivi che supportano tale livello di API e che comunque l'applicazione funzionerà anche per livelli di API inferiori fino a quello che è stato specificato come minimo.
Infine è possibile definire il livello di API massimo che un dispositivo dovrà avere per eseguire l'applicazione. Per fare questo l'attributo da utilizzare è il seguente:

```
android:maxSdkVersion
```

È comunque fondamentale sottolineare come si tratti di una informazione superflua in quanto, grazie alla compatibilità all'indietro delle versioni, il fatto che l'applicazione funzioni per la versione N garantisce che la stessa funzioni per tutte le versioni M >= N.

Configurazioni hardware e compatibilità

In precedenza abbiamo sottolineato come la gestione delle feature fosse un qualcosa di legato agli aspetti software di un particolare livello di API. Questo nel senso che, sebbene un dispositivo possa non avere un determinato componente hardware (come la videocamera o il relativo flash), comunque deve possedere tutte le API previste dal suo API Level. Le feature permettono di assicurarsi che vengano installate in un dispositivo solo le applicazioni che utilizzano le API che lo stesso è in grado di eseguire. Se un device non ha la videocamera non eseguirà mai applicazioni che ne richiedano l'utilizzo. Android comunque prevede un sistema simile anche per quelle funzionalità specificatamente hardware come la presenza di un tastierino numerico, di una trackball o altri sistemi di interazione. Tutto questo avviene attraverso la definizione di quelle che si chiamano `configuration` attraverso il seguente elemento XML da utilizzare all'interno dell'`AndroidManifest.xml`.

Listato 1.4 Elemento <uses-configuration/> per la definizione degli elementi HW supportati

```
<uses-configuration
    android:reqFiveWayNav=["true" | "false"]
    android:reqHardKeyboard=["true" | "false"]
    android:reqKeyboardType=["undefined" | "nokeys" | "qwerty" |"twelvekey"]
    android:reqNavigation=["undefined" | "nonav" | "dpad" | "trackball" |
    "wheel"]
    android:reqTouchScreen=["undefined" | "notouch" | "stylus" |"finger"] />
```

Si tratta di informazioni che quindi verranno utilizzate per filtrare quelle applicazioni che richiedono specifiche caratteristiche hardware. Nel caso in cui volessimo, per esempio, fare in modo che la nostra applicazione fosse utilizzata solo da dispositivi dotati di tastiera fisica e modalità touch, dovremmo definire nell'`AndroidManifest.xml` il seguente elemento::

```
<uses-configuration android:reqHardKeyboard="true" android:reqTouchScreen=finger"/>
```

Se poi volessimo estendere l'utilizzo anche ai dispositivi touch con pennino, dovremmo specificare due elementi ed esattamente:

```
<uses-configuration android:reqHardKeyboard="true" android:reqTouchScreen="finger"/>
<uses-configuration android:reqHardKeyboard="true" android:reqTouchScreen="stylus"/>
```

Si tratta quindi di informazioni utilizzate durante il processo di selezione delle applicazioni compatibili con un particolare tipo di device che possono essere utilizzate anche a runtime attraverso metodi della classe `PackageManager` del tipo:

```
public abstract List<PackageInfo> getInstalledPackages (int flags)
```

il quale permette di ottenere un elenco di oggetti di tipo `PackageInfo` che contengono informazioni relativamente ai package installati nel dispositivo che soddisfano i filtri specificati attraverso opportuni flag. La classe `PackageInfo` definisce quindi un attributo pubblico:

```
public ConfigurationInfo[] configPreferences;
```

che permette di accedere a un array di oggetti di tipo `ConfigInfo` che incapsulano le informazioni specificate attraverso l'elemento `<uses-configuration/>`. Si tratta di un meccanismo simile al precedente che permette di effettuare delle verifiche sulle configurazioni hardware e non su quelle software legate al particolare API Level.

Gestire schermi di dimensioni diverse

Come abbiamo più volte ricordato, una delle parti più importanti di un'applicazione mobile è sicuramente la User Interface la quale determina le diverse modalità di interazione da parte dell'utente. Oltre che l'aspetto puramente grafico dei componenti, peraltro molto importante, conta sicuramente il modo in cui gli stessi vengono posizionati nel display. Non tutti i dispositivi Android, anche se compatibili con uno stesso livello di API e dotati delle stesse feature hardware, sono necessariamente

dotati di display di uguale dimensione. Da queste considerazioni capiamo come sia di fondamentale importanza disporre di strumenti che permettano la realizzazione di applicazioni che in qualche modo si adattano al particolare display garantendo un ottimo livello di usabilità. Anche da questo punto di vista Android fa tesoro delle esperienze di altre tecnologie e in particolare modo dell'ambiente J2ME nel quale le UI possono essere definite attraverso API di alto livello e API di basso livello. Le API di alto livello sono quelle che permettono la realizzazione di interfacce il più possibile portabili in dispositivi diversi. Questo avviene attraverso la definizione di componenti astratti che vengono poi gestiti dalle specifiche implementazioni del MIDP. Il classico esempio è quello delle operazioni associate alla selezione di un soft key. La definizione di un `Command` di tipo `EXIT` permette, per esempio, di definire quello che è il comando di uscita dall'applicazione senza specificare però dove lo stesso verrà effettivamente posizionato. Sarà responsabilità della particolare implementazione MIDP mettere lo stesso comando nella posizione in cui lo stesso device mette comandi di quel tipo. Ecco che l'utente che intende uscire dall'applicazione MIDP eseguirà le stesse operazioni che gli permettono, nello stesso device, di uscire dalle altre applicazioni. Come detto si tratta di API che permettono di descrivere le UI in modo portabile ma al prezzo di grosse limitazioni nell'insieme di elementi disponibili. Per questo motivo esistono le API di basso livello le quali danno la possibilità di definire i diversi componenti con la precisione del singolo pixel nel display. Realizzare una UI utilizzando le API di basso livello non è cosa semplice se si pensa che i display possono avere dimensioni molto diverse tra loro. Il disegno dei componenti specificandone la posizione e le dimensioni in pixel rende la cosa infatti molto difficoltosa. Per questo motivo Android ha scelto una filosofia diversa che andiamo a descrivere e che, come vedremo, prevede che sia lo stesso ambiente a visualizzare l'interfaccia definita dallo sviluppatore gestendo in modo appropriato il posizionamento dei diversi componenti modificandone, eventualmente, le dimensioni. Il prezzo da pagare è comunque molto basso e consiste nel seguire, nella definizione delle UI, semplici regole che potremmo definire di buon senso.
Per una buona comprensione della filosofia adottata da Android è comunque bene dare alcune definizioni che possiamo riassumere nella Tabella 1.3.

Tabella 1.3 I concetti alla base delle UI in Android

Concetto	Descrizione
Screen size	È una grandezza indicativa delle dimensioni di un display e viene solitamente descritta in termini di lunghezza della diagonale in pollici (come le televisioni per intenderci).
Aspect ratio	Rappresenta una grandezza indicativa del rapporto tra la larghezza e l'altezza del display di un dispositivo.
Resolution	Indica il numero di pixel del display in orizzontale e verticale.
Density	Indica il numero di pixel per unità di lunghezza fisica.
Density Indipendent Pixel (dip)	Permette di rappresentare una dimensione in modo indipendente dalla densità.

La prima riguarda le dimensioni del display (screen size). Si tratta di un valore che viene solitamente specificato in termini di lunghezza, in pollici (inch), della diagonale

dello schermo. Come vedremo, Android non permetterà la gestione delle dimensioni dello schermo in termini di dimensioni assolute ma crea una generalizzazione che consiste nella classificazione di schermi grandi (large), normali (normal) e piccoli (small). Dalla versione 2.3 della piattaforma è stata aggiunta la dimensione grandissima (xlarge) probabilmente in preparazione di quelli che saranno i display supportati dalla versione 3.0 che sappiamo essere dedicata ai tablet. Lo sviluppatore, attraverso l'utilizzo di opportuni qualificatori, avrà comunque la possibilità di definire risorse diverse, e in particolare layout e dimensioni diverse, in corrispondenza di ciascuna di queste generalizzazioni. Sarà poi cura dell'ambiente scegliere quella corrispondente alle caratteristiche del display del dispositivo. Come vedremo, la definizione di versioni diverse di layout per ciascuna delle dimensioni possibili di un display non è sempre necessaria in quando Android stesso avrà la possibilità di applicare le opportune trasformazioni in modo automatico.

L'aspect ratio permette di fornire indicazioni su quello che è il rapporto tra la larghezza di un display rispetto alla sua altezza. Anche in questo caso è possibile utilizzare questa informazione come qualificatore e quindi definire layout diversi a seconda che il valore sia long o notlong. Possiamo comunque dire che si tratta di un qualificatore non molto utilizzato se non nella gestione di schermi in cui il rapporto tra larghezza e altezza sia sensibilmente più grande di quello che solitamente si ha nella generalizzazione base ovvero quella relativa a una dimensione `normal`.

Quando si parla di qualità di uno schermo si parla spesso di risoluzione la quale viene espressa in numero di pixel in orizzontale per il numero di pixel in verticale. Nonostante questo, non è comunque detto che la risoluzione di un display sia necessariamente legata all'aspect ratio. Tipici valori di risoluzione sono 320×480 o 240×320 ma si tratta di informazioni che comunque Android non considera come possibili qualificatori.

L'informazione più importante nella definizione delle dimensioni in Android è invece la densità ovvero il numero di pixel per unità di dimensione fisica. A parità di dimensioni fisiche, un display ad alta densità avrà un numero di pixel superiore rispetto a un altro a bassa densità. A livello intuitivo possiamo associare la densità a un modo per esprimere quanto piccoli e definiti siano i pixel utilizzati nella definizione di una qualche figura la quale apparirà meglio definita se la densità è alta rispetto al caso di densità bassa. Il concetto di densità è la principale ragione per cui le dimensioni in Android, per quanto possibile, non debbano mai essere espresse in pixel. Prendiamo per esempio due dispositivi con display con le stesse dimensioni fisiche ma con densità diversa: il primo bassa densità e il secondo alta densità. Se la densità nel primo è minore di quella del secondo, il numero di pixel contenuti nelle due dimensioni sarà sicuramente minore e quindi lo spazio occupato da ciascuno di essi sarà maggiore. Se esprimiamo quindi la dimensione di un bottone in pixel otterremo che la dimensione assoluta del bottone nel primo display sarà superiore a quella assoluta dello stesso bottone nel secondo display.

L'aspetto fondamentale riguarda il fatto che, anche per la densità, Android definisce quattro generalizzazioni per la classificazione di schermi con bassa (ldpi), media (mdpi), alta densità (hdpi) e, dalla versione 2.3, altissima densità (xhdpi). Si tratta di valori che possono essere utilizzati come qualificatori per cui, anche alla luce di quanto visto in precedenza, la piattaforma ci permetterebbe di realizzare delle risorse di layout o specializzare le immagini per ciascuna delle combinazioni di dimensioni, aspect ratio e densità.

Figura 1.3 Utilizzo di unità di misura diverse.

Questo significa che le possibilità di customizzazione sono 4×4×2 ovvero 32. Come vedremo nel successivo paragrafo, si tratta di un caso limite in quanto la maggior parte delle customizzazioni sono gestite in modo automatico dalla piattaforma Android. Per semplificare questo processo sarebbe quindi opportuno che le dimensioni fossero espresse in modo indipendente dalla densità del display. Questa unità di misura esiste e si chiama Density Independent Pixel (dip). Per dare un significato a questa grandezza definiamo anche che cosa significa dpi (density per inch) con cui la dip (indicata anche con dp) viene spesso confusa. Come dice l'acronimo, le dpi rappresentano un modo per indicare una densità ovvero un numero di pixel in una unità di misura fisica che in questo caso è il pollice (in inglese inch). Quella che viene indicata dalle specifiche Android come `baseline configuration`, rappresenta un insieme di dispositivi caratterizzati dall'avere un display `normal` e un densità `medium`. Ecco che 1 dp coincide con 1 pixel in uno schermo con densità 1 dpi. Ecco che se la densità dello schermo aumenta, 1 dp corrisponderà a una dimensione fisica maggiore.
Per dare una minima dimostrazione di quanto detto, abbiamo creato il progetto `TestScreenSize` il quale non fa altro che definire un layout con due bottoni le cui dimensioni sono state specificate in modo diverso. Il primo ha dimensioni di 50×100 pixel mentre il secondo 50×100 dp. Se visualizziamo i due pulsanti in un display classificato come di densità e dimensioni medie (associato alla sigla HVGA) otteniamo quello nella Figura 1.4 dove le dimensioni dei pulsanti sono uguali.

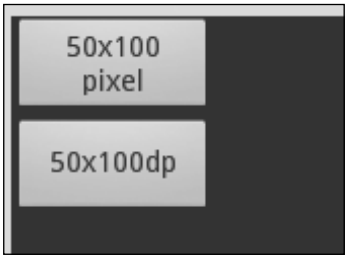

Figura 1.4 Le dp in un display mdpi e normal.

In questo caso, infatti, 1 dp coincide con 1 pixel. Se modifichiamo poi la risoluzione del display in una ad alta densità (per esempio WVGA800) otteniamo quello mostrato in Figura 1.5.

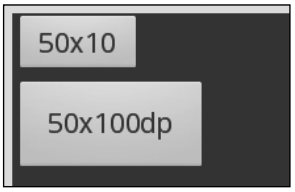

Figura 1.5 Pixel e db in un display hdpi e normal.

In questo caso il bottone con le dimensioni specificate in pixel ha dimensioni fisiche ridotte rispetto al bottone le cui dimensioni sono state specificate in dp. Infatti, in un display ad alta densità il numero di pixel per unità di lunghezza è maggiore. L'aspetto importante non riguarda comunque la dimensione fisica dei diversi elementi nel display quanto il mantenimento delle loro proporzioni. Per questo motivo è sempre bene utilizzare dimensioni relative anche se, come vedremo successivamente, anche nel caso dell'utilizzo delle dimensioni in pixel, l'ambiente ci darà una mano.

Gestione automatica delle dimensioni

Come abbiamo accennato in precedenza, Android ci fornisce una serie di aiuti per fare in modo che i componenti visuali utilizzati come interfaccia delle `Activity`, si adattino nel modo migliore a quelle che sono le caratteristiche di dimensione e densità dei diversi display. Nel caso in cui questo comportamento non fosse sufficiente per la particolare applicazione, esistono comunque dei punti di intervento che prevedono l'utilizzo di qualificatori nella definizione delle risorse, oppure l'impostazione di alcuni attributi dell'elemento `<supports-screens/>` nell'`AndroidManifest.xml`.
Il primo di questi strumenti si applica ovviamente alle risorse attraverso l'utilizzo dei qualificatori relativi alla densità e/o alle dimensioni. Per esempio, se volessimo specificare una immagine per ciascuna densità sarà sufficiente creare 4 cartelle `Drawable` all'interno delle quali inserire la corrispondente versione dell'immagine. Se il nome dei file delle immagini è `custom.png`, dovremo definire le seguenti 4 cartelle:

```
/res/drawable-ldpi
/res/drawable-mdpi
/res/drawable-hdpi
/res/drawable-xhdpi
```

e inserire in ciascuna di esse la corrispondente versione del file `custom.png` con la risoluzione voluta. In questo caso, il sistema non eseguirebbe, per quella risorsa, alcuna trasformazione automatica ma visualizzerebbe l'immagine così come è stata definita. Nel caso in cui non si facesse uso di questi qualificatori, il sistema prenderebbe come riferimento la risorsa indicata di default (nella cartella associata alla densità media o quella senza qualificatori) e ne farebbe un resize automatico in relazione a come il display del

dispositivo è stato classificato. Se per esempio il display è classificato come ad alta densità, le dimensioni dell'immagine verrebbero moltiplicate per 1.5 in modo da avere le stesse dimensioni fisiche che la stessa avrebbe in un display a densità media. Il lettore potrebbe chiedersi l'origine del valore 1.5 il quale è semplicemente il rapporto tra la densità che Android considera come alta, ovvero 240, e quella considerata media ovvero 160. Ecco che 240/160 = 1.5. Nel caso di un display a densità altissima, e quindi con un valore di 320 dpi, il fattore moltiplicativo sarebbe 2.
Il secondo punto di configurazione è quello relativo all'utilizzo dell'elemento `<supports-screens/>` il quale permette di fare due tipi di impostazioni relativamente a dimensioni e densità che vengono considerate da Android ortogonali l'una rispetto all'altra e quindi indipendenti.
Relativamente alle dimensioni è possibile utilizzare gli attributi in tabella i quali vengono tenuti in considerazione sia dall'Android Market che dal sistema a runtime.

Tabella 1.4 Attributi di <support-screens/>

Attributi di `<supports-screens>`	**Descrizione**
`android:smallScreens`	Permette di specificare se l'applicazione supporta schermi di dimensione inferiore a quella classificata come `normal`.
`android:normalScreens`	Permette di specificare se l'applicazione supporta schermi di dimensione uguale a quella classificata come `normal`.
`android:largeScreens`	Permette di specificare se l'applicazione supporta schermi di dimensione superiore a quella denominata come `normal`.
`android:xlargeScreens`	Dalla versione 2.3 della piattaforma è possibile specificare se un'applicazione supporta schermi di dimensione classificata come `xlarge`.

Come detto, si tratta di attributi che indicano quelle che sono le dimensioni dello schermo gestite dall'applicazione. Un valore `true` di questi attributi indica che l'applicazione è in grado di gestire il posizionamento e ridimensionamento dei componenti per quella particolare tipologia di display. Questo significa che il sistema non agirà in alcun modo lasciando all'applicazione tutta la responsabilità. Un valore `false` indica invece che l'applicazione non è in grado di gestire display della corrispondente dimensione per cui delega al sistema il quale farà in modo di produrre interfacce coerenti rispetto a quelle considerate di default. Si tratta di informazioni che vengono utilizzate anche dall'`AndroidMarket` per decidere se una particolare applicazione può essere eseguita su un dispositivo di dimensione data. La regola di selezione non è comunque drastica nel senso che non è detto che un'applicazione che supporta solamente un tipo di dimensione dello schermo venga scaricata solamente da dispositivi di quelle dimensioni. La regola in tale senso dice che un'applicazione può essere eseguita da un dispositivo se questa supporta almeno una delle dimensioni uguali o inferiori a quelle del display del dispositivo stesso. Questo significa che se un'applicazione fornisce un valore `true` solamente per l'attributo `android:normalScreens` e false per tutti gli altri, essa potrà essere eseguita da tutti i dispositivi con schermo di dimensione `normal` o superiore. Questo perché il sistema provvede in modo automatico al ridimensionamento dei diversi componenti. La stessa applicazione non verrà invece eseguita in quei dispositivi con display classificato come `small`. In sintesi il sistema provvede in modo automatico solamente a ridimensionamenti verso l'alto e non verso il basso.

È importante sottolineare come questi attributi facciano riferimento alle dimensioni dello schermo e non alla loro densità.

Utilizzo di attributi della 2.3

Da quanto detto sopra la possibilità di utilizzare o meno l'attributo `xlargeScreens` dipende dal valore specificato dall'attributo `android:targetSdkVersion`.

In tale senso assume fondamentale importanza il valore dell'attributo `android:anyDensity` il quale ci permette di informare il sistema del fatto che la nostra applicazione gestisca in modo autonomo le differenze di densità tra i vari display oppure no. Il valore di questo attributo influenza quindi anche il comportamento del sistema nei confronti delle varie risorse.

Ridimensionamento dei Drawable

È bene precisare come l'identificazione e/o resize dei `Drawable` a seconda dei particolari qualificatori avvenga indipendentemente dalle configurazioni relative alle dimensioni e densità che stiamo descrivendo in questo paragrafo.

Nel caso in cui un'applicazione definisse un valore `false` per questo attributo, il sistema si attiverebbe per fare in modo che le varie dimensioni assolute specificate vengano comunque visualizzate in modo corretto nei display con densità diverse. Per verificare il comportamento di questo attributo supponiamo di creare un semplicissimo layout che contiene un pulsante le cui dimensioni sono specificate in pixel e in particolare di 160×160 pixel. Creiamo quindi due AndroidVirtual Device (AVD) dove il primo utilizza un display HVGA di densità media mentre il secondo un display di tipo WVGA800 di alta densità. Si tratta di due schermi di dimensioni diverse ma comunque classificati entrambi come `normal`. Inizialmente il valore dell'attributo `android:anyDensity` è a `false` ovvero stiamo indicando ai dispositivi che la nostra applicazione non gestisce la variazione delle densità e quindi ci affidiamo al sistema il quale produrrà il risultato della Figura 1.6.

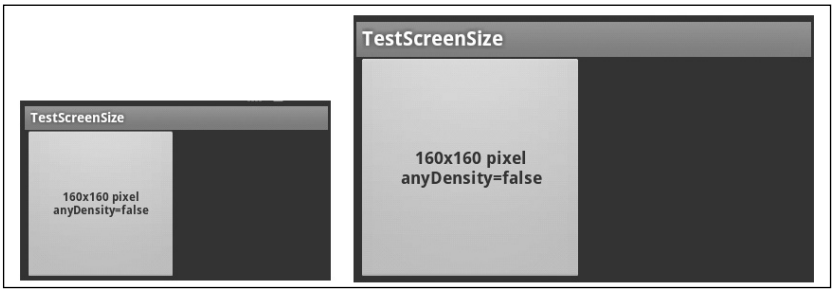

Figura 1.6 Utilizzo del valore false per l'attributo android:anyDensity.

Sulla sinistra abbiamo il display di tipo HVGA mentre sulla destra quello di tipo WVGA800. Notiamo come avendo dichiarato di non essere in grado di gestire

autonomamente le dimensioni del pulsante, il sistema ha gestito le proporzioni in modo automatico. Sebbene il primo display abbia dimensioni di 320×480 mentre il secondo abbia dimensioni di 480×800, comunque il pulsante occupa metà larghezza mantenendo quindi le giuste dimensioni. L'aspetto interessante riguarda il fatto che l'applicazione sia stata creata considerando valide le dimensioni caratteristiche di uno schermo `normal` con densità media. Questo significa che per lo sviluppatore il pulsante ha dimensioni 160×160 anche nel caso in cui venga eseguito nel display WVGA800. È il sistema che fa in modo che, per esempio, il punto di coordinate (100,100) venga mappato in modo completamente trasparente sul punto (150,150) del secondo display. Ovviamente abbiamo testato il comportamento verso una densità maggiore ma lo stesso si poteva applicare a densità inferiore. Un valore `false` dell'attributo `android:anyDensity` è quello che si ha nel caso di applicazioni realizzate con la versione dell'SDK 1.5 o precedenti per le quali le considerazioni viste non erano ancora valide.
È bene precisare che quanto visto è valido nel caso in cui si specificassero le dimensioni in pixel come fatto nel precedente esempio.

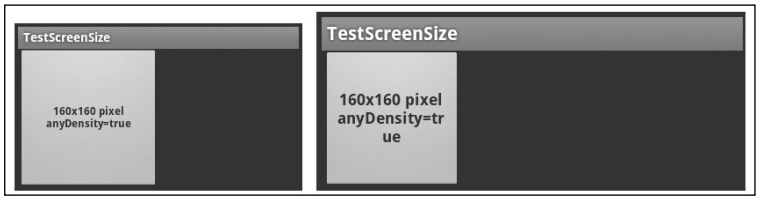

Figura 1.7 Utilizzo del valore true per android:anyDensity con le dimensioni in pixel.

Supponiamo infatti di utilizzare il valore `true` per l'attributo `android:anyDensity` informando quindi il sistema che la nostra applicazione gestisce le dimensioni in modo autonomo. Lasciando le dimensioni in pixel ed eseguendo nuovamente l'applicazione nei due AVD otterremo ora quello della Figura 1.7. Ora notiamo come non vengano più mantenute le proporzioni in quanto nel secondo display le dimensioni sono in minori avendo lo stesso infatti densità maggiore. Impostando a `true` il valore dell'attributo `android:anyDensity` ci siamo presi la responsabilità di gestire autonomamente le dimensioni per cui quello che dovremo fare è specificare tali dimensioni in modo indipendente dalla densità e quindi non più come pixel ma come dp. Modificando quindi le dimensioni del pulsante da 160×160 pixel in 160×160 dp otteniamo quello della Figura 1.8 ovvero quanto ottenuto nel primo caso.

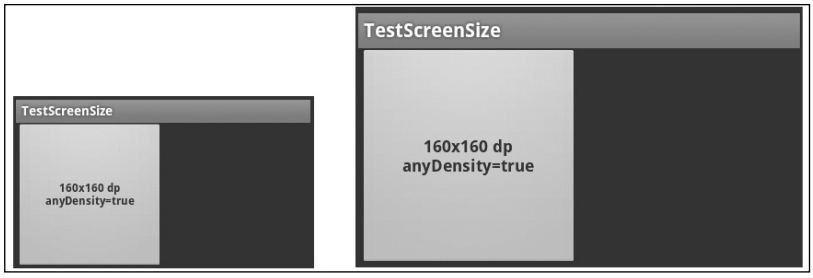

Figura 1.8 Utilizzo delle dimensioni con valori indipendenti dalla densità.

Calcolo esplicito delle dimensioni

Nel caso in cui l'applicazione si prendesse la responsabilità di gestire le dimensioni dei componenti per le diverse densità, è possibile utilizzare delle API che permettono di convertire i pixel in dp e viceversa. A tale proposito ci viene in aiuto la classe `DisplayMetrics` del package `android.util` la quale permette di ottenere le diverse informazioni sul display del dispositivo che sta eseguendo una particolare applicazione. Per ottenere il riferimento a questo oggetto è sufficiente eseguire le seguenti righe di codice come possiamo vedere nel progetto `DisplayMetricsApp` disponibile con il presente testo.

Listato 1.8 Riferimento all'oggetto DisplayMetrics

```
DisplayMetrics metrics = new DisplayMetrics();
getWindowManager().getDefaultDisplay().getMetrics(metrics);
```

Attraverso il metodo `getMetrics()` dell'oggetto di tipo `Display` ottenuto dal `WindowManager` con la modalità `getDefaultDisplay()`, è infatti possibile valorizzare le proprietà dell'oggetto `DisplayMetrics` passato come parametro. Un modo analogo e alternativo per ottenere lo stesso riferimento è il seguente:

Listato 1.9 Riferimento all'oggetto DisplayMetrics dalle risorse

```
DisplayMetrics metrics = getResources().getDisplayMetrics();
```

Eseguendo l'applicazione in un dispositivo classificato `normal` come quello con display HVGA del precedente esempio, otterremo quello mostrato in Figura 1.9.

Figura 1.9 Dati di un display con densità media e dimensione normal.

Possiamo osservare come il valore di densità ottenuto dalla proprietà *density* rappresenti il valore da utilizzare per le varie conversioni nelle dimensioni in pixel.
Se eseguiamo infatti la stessa applicazione con il secondo AVD dell'esempio precedente ovvero quello con schermo WVGA800 otteniamo quello della Figura 1.10 dove tale valore è ora di 1.5.

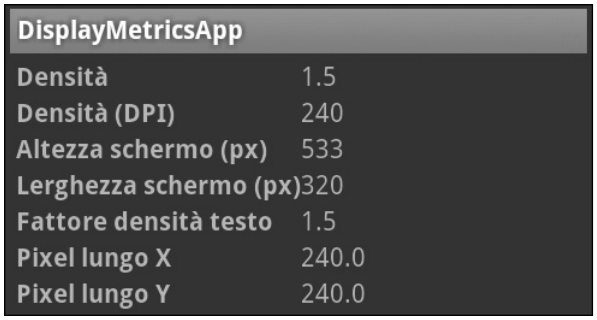

Figura 1.10 Dati di un display con densità alta e dimensione normal.

Questo significa che se volessimo convertire una dimensione in dp nella corrispondente dimensione in pixel sarà sufficiente eseguire la seguente operazione:

```
pixel = getContext().getResources().getDisplayMetrics().density * dp
```

Ecco che 160 dp corrispondono, in un display con density 1.0, a 160 pixel mentre corrispondono a 240 per un display con un display a densità alta e quindi con valore 1.5 della proprietà density dell'oggetto `DisplayMetrics`.
Concludiamo il paragrafo osservando come il sistema disponga della classe `Display` del package `android.view` a cui si ottiene un riferimento attraverso il metodo `getDefaultDisplay()` del `WindowManager`. Si tratta di una classe molto interessante che, oltre alle informazioni già viste, ci permette, per esempio, di conoscere la frequenza di aggiornamento del display attraverso il metodo:

```
public float getRefreshRate ()
```

in termini di frame per secondo. È un metodo che può essere molto utile nel valutare le prestazioni delle nostre applicazioni e l'impatto che queste hanno sulle capacità del dispositivo. Altra informazione utile è quella relativa alla posizione del dispositivo che prende il nome di `rotation` al posto della deprecata `orientation`. Si tratta di un valore che si ottiene attraverso il metodo:

```
public int getRotation ()
```

e che può ritornare i valori corrispondenti a una serie di costanti della classe `Surface` tra cui `Surface.ROTATION_0`, `Surface.ROTATION_180` e altri ancora di ovvio significato che il lettore potrà consultare nella documentazione ufficiale. Ovviamente si tratta di misure relative alla normale posizione del dispositivo che potrebbe comunque dipendere dallo stesso. Per uno smartphone la posizione normale è, per esempio, quella portrait. Altra informazione di basso livello riguarda infine il tipo di formato utilizzato per la rappresentazione dei pixel, informazione che si può ottenere attraverso il metodo:

```
public int getPixelFormat ()
```

e che ritorna un valore corrispondente alla relativa costante della classe `PixelFormat` dove alcuni possibili valori sono `RGB_332`, `L_8` e altri ancora.

Conclusioni

In questo primo capitolo abbiamo descritto gli strumenti che la piattaforma Android ci mette a disposizione per risolvere il problema della frammentazione dovuto alla presenza di moltissimi dispositivi diversi sia nelle caratteristiche hardware che software. Abbiamo visto il concetto di feature e di come queste vengano utilizzate dall'Android Market per fare in modo che i dispositivi eseguano solamente le applicazioni a essi compatibili. Abbiamo poi esaminato uno degli aspetti più importanti ovvero quello della configurabilità delle interfacce in relazione alle diverse tipologie di display sia in relazione alle loro dimensioni che alle loro densità.