Creare App per Android

di
Thomas Brivio

ANDROID

CREARE APP PER ANDROID

di

Thomas Brivio

Sommario

Capitolo 1 – Introduzione: perchè Android

Android sta dilagando. Non è più solo questione di smartphone o tablet. Si sta imponendo come sistema operativo in grado di animare qualsiasi dispositivo più o meno mobile tanto da apparire, in prospettiva non troppo futuristica, una presenza sempre più costante nel nostro quotidiano.

Gli è stata attribuita – e probabilmente a ragione – la più veloce diffusione mai vista per un sistema operativo mobile. Ma quali sono i fattori di questo successo e soprattutto perchè tutto ciò sta capitando ad Android? Le motivazioni ipotizzate sono varie e di

varia natura. Gran parte del merito è stato attribuito alle sue radici ben salde nel mondo open source. Android, infatti, è figlio di Linux, e ha attirato l'interesse di tanti sviluppatori che per anni si sono stretti intorno ai grandi bacini del software libero ed accoglie in sé tutto il meglio di quanto è stato ideato per supportare lo sviluppo del web, desktop e mobile sia in termini di pattern progettuali che di librerie software.

Eppure architettura del sistema ed open source sono aspetti che interessano molto una platea fortemente tecnica come programmatori ed ingegneri informatici. Per decretare un successo tanto ampio è necessario che ci sia anche un **forte riscontro di pubblico**.

Sicuramente un qualcosa che ha contraddistinto questo sistema è stata la sua adozione da parte di dispositivi molto diversi tra loro, non solo per tipologia – smartphone piuttosto che tablet – ma soprattutto per fasce di prezzo, da poche decine di euro fino a cifre piuttosto sgnificative.

Ciò ne ha permesso una **diffusione molto diverficata trasversalmente** alle diverse categorie sociali, ma ha causato di riflesso una frammentazione notevole dello scenario applicativo costringendo gli sviluppatori ad una particolare cura degli aspetti di adattamento alle caratteristiche del dispositivo ospite. Proprio in questo, Android ha dimostrato la sua grande modernità

offrendo tutto il supporto necessario per permettere all'applicazione in esecuzione di adegursi ad ogni circostanza.

Questa guida è dedicata allo sviluppo di applicativi Android. Nei prossimi capitoli verrà utilizzata una programmazione in linguaggio Java per realizzare applicazioni in cui non manca niente. Si creeranno interfacce utente interattive, dinamiche e graficamente piacevoli. Si avranno gli strumenti per gestire dati e avviare attività di rete in modo che le proprie app prendano da subito parte a quel grande laboratorio di idee che è Internet e tutti i servizi che ne fanno parte. Ed inoltre si avrà a disposizione multimedialità, funzionalità

hardware, comunicazione.

Per iniziare, Android richiede nozioni di programmazione Java, passione, curiosità e nulla più. Infatti gli strumenti che si renderanno necessari, come si vedrà presto, sono totalmente gratuiti. Quindi non ci sono scuse per non cominciare.

Capitolo 2 – L'SDK e l'ambiente di sviluppo

Per iniziare a programmare su Android è necessario munirsi innanzitutto dei necessari strumenti software, tutti velocemente reperibili in Internet a costo zero. Necessari sono:

- un **JDK**, il kit di sviluppo per la tradizionale programmazione Java, visto che questa è la tecnologia con cui realizzeremo i nostri programmi;

- un **IDE** (ambiente di sviluppo integrato) che includa possibilmente tutti gli strumenti necessari al programmatore. Ne verranno presentati due, le alternative più comuni ed entrambe valide: **Eclipse** ed **Android Studio**.

Tra gli strumenti appena citati, di cui a breve verranno illustrate le fasi di download ed installazione, non è stato nominato un elemento fondamentale che merita, però, una menzione speciale: l'**Android SDK**. Questo è il vero

pacchetto di strumenti che ci permetterà di vedere realizzati i nostri programmi per Android. Nonostante l'importanza fondamentale rivestita, il suo utilizzo, inizialmente, può passare un po' inosservato visto che normalmente viene scaricato insieme agli IDE più comuni, Eclipse e Android Studio. Per questo non mancherò di sottolinearne sin da ora la sua struttura e le funzionalità che ne fanno parte.

Iniziamo, se non lo si è già installato, ad insallare Java. È necessario recarsi presso il sito Oracle e scaricarne una versione per il proprio sistema operativo, specificando non solo la tipologia – Windows, Linux, Mac OS o Solaris - ma anche la versione, 32 o,

meglio se se ne ha la possibilità, 64 bit. Una volta eseguito lo scaricamento del pacchetto se ne procede all'installazione che non presenta grandi difficoltà in alcuno dei sistemi per cui è disponibile.

Maggiore interesse riveste la **scelta dell'IDE**. Come detto, le due alternative più praticabili al momento sono Eclipse e Android Studio.

ECLIPSE

Eclipse è uno strumento gratuito e molto flessibile, ben noto da tanti anni a varie comunità di sviluppatori. In particolare, la sua natura modulare l'ha reso molto ricco di funzionalità mediante vari plugin installabili al suo interno, oltre che utilizzabile nella programmazione con vari linguaggi, primo tra tutti Java, ma anche C/C++, PHP ed altro ancora. Anche nel caso di Android esiste un plugin, chiamato strong-ADT (**A**ndroid **D**evelopement **T**ools).

La sua installazione consiste nei seguenti passi:

1. **Download di Eclipse**: è necessario ottenere una versione di Eclipse pari o superiore alla 3.7.2 (chiamata *Indigo*): la versione per Java Developers andrà benissimo. L'IDE dovrebbe avere al suo interno il plugin JDT; verificare la sua presenza, nelle versioni recenti, risulta superfluo trattandosi ormai di una caratteristica di serie;

2. **Installazione del plugin ADT**: Eclipse viene fornito in versione

stand-alone; in altre parole, esso non va installato, ma è sufficiente decomprimerlo ed eseguirlo. Una volta avviato, per installare ADT, si selezioni la voce *Install New Software* nel menu *Help*. La finestra che si apre presenta un pulsante *Add*, tramite cui si può inserire l'indirizzo da cui Eclipse provvederà per il **Installazione del plugin ADT**: Eclipse viene fornito in versione *stand-*

alone; in altre parole, esso non va installato, ma è sufficiente decomprimerlo ed eseguirlo. Una volta avviato, per installare ADT, si selezioni la voce *Install New Software* nel menu *Help*. La finestra che si apre presenta un pulsante *Add*, tramite cui si può inserire l'indirizzo da cui Eclipse provvederà per il L'URL richiesto è il seguente: http://dl-ssl.google.com/android/ec
Dopo la connessione alla

sorgente, nella finestra *Available Software*, apparirà il componente *Developer Tools*: si dovrà spuntare la checkbox relativa ad esso e proseguire con l'installazione tramite pulsante *Next*. Successivamente, appena terminerà il download del componente, sarà necessario accettare la licenza d'uso proposta;

3. **Download dell'Android SDK**: il plugin appena installato permetterà di avere a

disposizione voci di menu relative allo sviluppo Android all'interno di Eclipse. L'ultimo passo consiste nel collegamento dell'IDE ad un **Android SDK**. Quest'ultimo, qualora non lo si possieda già nella propria macchina, può essere scaricato dal sito ufficiale di Android. Lo si dovrà scompattare sul nostro file system (ad esempio può essere posto all'interno della directory di Eclipse), e

specificare il path della cartella nelle *Preferences* di Eclipse: selezionare la voce di menu *Window* - *Preferences* e, nella sezione *Android* dell'interfaccia che appare, inserire il path del proprio SDK nella casella di testo *SDK Location*.

Android Studio

Un'alternativa giovane ma molto promettente è Android Studio, sponsorizzato direttamente da Google e pensato appositamente per Android. Gli elementi di Android Studio che spiccano maggiormente sono:

- l'utilizzo di Gradle come strumento di build automation, atto quindi ad accompagnare lo sviluppatore nelle fasi di build, sviluppo, test e pubblicazione della propria app;

- disponibilità di un gran numero di template per la realizzazione di applicazioni già in linea con i più comuni pattern progettuali;
- editor grafico per la realizzazione di layout, molto pratico e dotato di un ottimo strumento di anteprima in grado di mostrare l'aspetto finale dell'interfaccia che si sta realizzando in una molteplicità di configurazioni (tablet e smartphone di vario tipo).

Android SDK

Ultima nota, diamo uno sguardo più ravvicinato all'Android SDK. Come detto, la sua installazione è inclusa in quella di Android Studio o Eclipse. Un aspetto molto importante è che questo SDK è costituito da molti strumenti – programmi, emulatori, piattaforme per ogni versione di Android e molto altro – la cui composizione non è immutabile ma viene gestita tramite il programma Android SDK Manager, avviabile sia da Eclipse che da Android Studio. Grazie al Manager, il programmatore potrà profilare le piattaforme e gli strumenti presenti nel SDK nella maniera più congeniale al proprio lavoro.

Un paio di download, qualche click per installare e scompattare: indipendentemente dal sistema operativo del proprio PC, l'ambiente per lo sviluppo su Android non necessità di grandi operazioni per essere pronto all'utilizzo e alla realizzazione della prima app.

Capitolo 3 – Alternative allo sviluppo nativo

L'approccio alla programmazione Android appare del tutto agevole soprattutto grazie a due aspetti fondamentali già evidenziati: strumenti del tutto gratuiti e semplicità nell'apprestamento dell'ambiente di sviluppo. Ed in effetti è così. A volte però l'appassionato di tecnologia che si avvicina a questo mondo rischia di scoraggiarsi facilmente.

Ammesso che si possegga le skill necessarie del linguaggio Java, ci si **accorge presto che un'infarinatura di sintassi spesso non è sufficiente**. Per sfruttare degnamente le possibilità

offerte dal framework è necessario essere dei buoni programmatori, consci delle principali problematiche da tenere sott'occhio in uno sviluppo professionale: ottimizzazione delle prestazioni, salvaguardia delle risorse a disposizione e via dicendo.

Ma allora che fare se non ci si sente attratti dalla programmazione di questo tipo, cosiddetta *nativa*, e non si vuole comunque rinunciare all'idea di vedere pubblicate le proprie app?

Alternative ce ne sono e consistono in strumenti – comunque validi – per lo sviluppo **non nativo**, dall'approccio più visuale e spesso familiare a chi proviene dal web design.

Eccone alcuni:

- Apache Cordova e Steroids scaturiscono dal know-how nello sviluppo per Internet. In effetti: se CSS ha reso il Web elegante, Javascript gli ha dato vitalità e HTML5 l'ha rivoluzionato, perché non riproporre questi strumenti nello sviluppo mobile? Cordova è la versione open source del progetto PhoneGap e serve a realizzare le cosiddette app ibride con

un'interfaccia realizzata in modalità web ma in grado di interfacciarsi con il sistema operativo mobile mediante un vasto numero di API. Steroids nasce per superare alcuni limiti riscontrati in PhoneGap ma senza "reinventare la ruota", si basa infatti su Cordova ma approfondisce il legame con lo strato nativo;

- Corona SDK è un ambiente particolarmente versato alla gestione dell'interazione come per

i videogiochi. Creato dai Corona Labs non necessità di alcuna conoscenza del linguaggio Java e propone come alternativa lo scripting in LUA, un formalismo dall'approccio molto semplice che permette di personalizzare maggiormente le proprie applicazioni;

- Unity è il più diffuso motore per videogiochi del mondo. Il primato che gli spetta è pienamente meritato in quanto

combina editor visuale di altissimo livello ma anche programmazione in tecnologie ad oggetti avanzate come C# oltre che gestione di altri aspetti fondamentali per i videogiochi come grafica, animazione e fisica;

- AppInventor è probabilmente la possibilità più abbordabile per l'appassionato di tecnologia a digiuno di programmazione e che desidera qualche

risultato piuttosto rapido. E' stato creato dai Google Labs come strumento per la rapida modellazione di app Android ed è stato successivamente ceduto al prestigioso M.I.T.. Intuitività e semplicità le sue chiavi di lettura principali.

Nativo sì, nativo no. **Qual è l'approccio migliore?** Sicuramente entrambi hanno i loro pro e contro. Mentre da un lato il nativo offre la possibilità di una gestione totale del dispositivo senza la paura di trovare

limiti, d'altra parte richiede spesso una programmazione molto professionale e si concentra esclusivamente su una piattaforma impedendo un'agile riciclo dei propri sforzi su altri mercati del mobile.

Il non-nativo – anche se è impossibile generalizzare data la diversità degli ambienti appena citati – offre vantaggi vari, ascrivibili a volte ad una minore necessità di programmare e molto spesso alla possibilità di creare applicazioni *cross-platform* distribuibili su sistemi operativi diversi.

Capitolo 4 – Gli elementi e il funzionamento di base di un'applicazione

Ogni applicazione Android, indipendentemente dalla finalità che si prefigge, affida le sue funzionalità a quattro tipi di componenti. Si tratta di **Activity**, **Service**, **Content Provider** e **BroadcastReceiver** ed esistono affinchè la nostra applicazione possa integrarsi alla perfezione nell'ecosistema Android.

Prima di addentrarci nella spiegazione di ognuna di esse, è utile concentrarsi un attimo su due principi ispiratori che, tra gli altri, sono alla base della maggior parte delle scelte

progettuali operate dai creatori di Android. Tenerli a mente ci permetterà di comprendere meglio ciò che del sistema verrà illustrato nei prossimi capitoli:

- **la salvaguardia delle risorse**: essendo progettato per sistemi embedded, storicamente dotati di poche risorse di memoria, Android ha avuto sin da subito uno spirito parsimonioso. Vedremo che, senza far perdere fluidità alla *user-experience,*

Android è particolarmente bravo nel distruggere e ricreare parti dell'applicazione in maniera del tutto impercettibile all'utente. Chi dovrà fronteggiare questo atteggiamento sarà il programmatore, ovviamente. Per fortuna, ciò non costerà grandi fatiche ma solo particolare cura nel prendere determinati accorgimenti da applicare con la necessaria consapevolezza. Vale la

pena sottolineare che quando si parla di esiguità di risorse in Android, l'obiezione mossa più comunemente ruota attorno alla recente commercializzazione di smartphone che possono contare su 2 GB di memoria RAM. Ciò è vero ma non bisogna dimenticare che Android si propone lo scopo di animare *qualunque* dispositivo in cui riesca a vivere. Il sistema vincerà quindi la sfida di sopravvivenza solo se

saprà adattarsi anche a contesti che offrono condizioni molto più disagiate di quelle che può prospettare un nuovissimo device Samsung;

- **sicurezza**: Android è figlio di Linux, come già ricordato, quindi ha nel DNA la ricerca della stabilità. Ogni applicazione è un utente a sé stante e vive in un proprio processo in cui viene allocata una nuova istanza della virtual machine, ciò per evitare

che il crash di un'applicazione propaghi instabilità alle altre app in esecuzione. Questa forma di "isolamento" viene riflessa anche sulla memoria di massa in quanto ogni applicazione ha un suo spazio in cui lavorare e custodire i propri dati. In merito, **è assolutamente sconsigliata, per non dire vietata, qualsiasi pratica che porti un'app ad invadere lo spazio riservato ad un'altra**. Nonostante ciò le nostre

applicazioni non sono costrette a vivere in assenza di comunicazione tra loro, anzi Android favorisce un dialogo "sano" tra di esse mettendo a disposizione meccanismi agevoli per la condivisione di contenuti e funzionalità tra componenti del sistema.

È arrivato il momento quindi di presentare più da vicino i blocchi costitutivi di un'applicazione.

Le fondamenta di Android

Un'**Activity** è un'interfaccia utente. Ogni volta che si usa un'app generalmente si interagisce con una o più "pagine" mediante le quali si consultano dati o si immettono *input*. Ovviamente la realizzazione di Activity è il punto di partenza di ogni corso di programmazione Android visto che è il componente con cui l'utente ha il contatto più diretto.

U n **Service** svolge un ruolo, se vogliamo, opposto all'Activity. Infatti rappresenta un lavoro – generalmente lungo e continuato – che viene svolto interamente in background senza bisogno di interazione diretta con l'utente. I

Service hanno un'importanza basilare nella programmazione proprio perchè spesso preparano i dati che le activity devono mostrare all'utente permettendo una reattività maggiore nel momento della visualizzazione.

Un **Content Provider** nasce con lo scopo della condivisione di dati tra applicazioni. La sua finalità richiama quel principio di sicurezza dell'applicazione di cui si è trattato poco fa. Questi componenti permettono di condividere, nell'ambito del sistema, contenuti custoditi in un database, su file o reperibili mediante accessi in Rete. Tali contenuti potranno essere usati da altre applicazioni senza invadere lo spazio di memoria ma stabilendo quel

dialogo “sano” cui si è accennato

Un **Broadcast Receiver** è un componente che reagisce ad un invio di messaggi a livello di sistema – appunto in *broadcast* – con cui Android notifica l’avvenimento di un determinato evento, ad esempio l’arrivo di un SMS o di una chiamata o sollecita l’esecuzione di azioni. Questi componenti come si può immaginare sono particolarmente utili per la gestione istantanea di determinate circostanze speciali.

Molto importante ricordare che una componente può attivarne un’altra mediante apposite invocazioni di sistema. Questa *intenzione* viene codificata con un **Intent** utilizzabile

come normale classe Java ma che sottintende un potentissimo strumento di comunicazione di Android. Anche degli Intent faremo uso sin dai prossimi capitoli.

Capitolo 5 – Il ciclo di vita di un'app Android

Android sa che il fattore fondamentale della sopravvivenza di un sistema mobile è la **corretta gestione delle risorse**. Pensiamo ad uno smartphone: è un dispositivo che fa una vita difficile al giorno d'oggi. Non solo si occupa di chiamate ed SMS, ma offre pagine web, giochi, comunicazione sui "social" per molto tempo ogni giorno. Inoltre, capita sempre più spesso che non venga mai spento impedendo così una fase molto comune nella vita dei PC: l'arresto del sistema con conseguente liberazione della memoria e pulizia di risorse temporanee assegnate.

Android farà in modo di **tenere in vita ogni processo il più a lungo possibile**. Ciò non toglie che in alcune circostanze ed in base alle risorse hardware a disposizione, il sistema operativo si troverà nella necessità di dover liberare memoria abbattendo processi.

Sì ma: quale processo abbattere? La discriminante è quanto un'applicazione, candidata all'eliminazione, sia importante per la **user experience**. Maggiore sarà l'importanza riconosciuta, minori saranno le probabilità che venga arrestata.

Così facendo Android tenterà di raggiungere il suo duplice scopo:

preservare il sistema e salvaguardare l'utente.

I processi possono essere classificati, in ordine di importanza decrescente, come:

1. **Processi in "foreground"**: sono quelli che interagiscono direttamente o indirettamente con l'utente. Stiamo parlando delle applicazioni che, ad esempio, contengono l'Activity attualmente utilizzata o i Service ad essa collegati. Questi

sono i processi che Android tenterà di preservare maggiormente.
Importante notare che, comunque, anche le applicazioni in foreground non sono del tutto al sicuro. Se ad esempio il sistema non disponesse di risorse sufficienti a mantenerli tutti in vita, si troverebbe costretto ad arrestarne qualcuno;

2. **Processi visibili**: non sono importanti come quelli in foreground ma

vengono anch'essi grandemente tutelati da Android. Infatti, avendo componenti ancora visibili all'utente anche se non vi interagiscono più, svolgono comunque un ruolo particolarmente critico. Anche in questo caso si tratta di Activity visibili e Service ad esse collegati;

3. **Processi "service"**: contengono dei service in esecuzione che generalmente svolgono lavori molto utili all'utente anche se non

direttamente collegati con ciò che egli vede nel display. Il loro livello di priorità può essere considerato medio: importanti sì ma non tanto quanto i processi di cui ai precedenti due punti;

4. **Processi in "background"**: contengono activity non più visibili all'utente. Questa è una categoria solitamente molto affollata composta dal gran numero di applicazioni che l'utente

ha usato e messo poi in disparte, ad esempio premendo il tasto Home. Non sono considerati molto importanti e sono dei buoni candidati all'eliminazione in caso di scarsità di risorse;

5. **Processi "empty"**: sono praticamente vuoti nel senso che non hanno alcuna componente di sistema attiva. Vengono conservati solo per motivi di cache, per velocizzare la loro riattivazione qualora si rendesse necessaria.

Come ovvio, sono i candidati "numero 1" all'eliminazione da parte del sistema operativo.

Quando, nel corso della guida, esamineremo in dettaglio la realizzazione delle varie componenti di sistema – Activity, Service, ContentProvider e BroadcastReceiver come accennato nel capitolo precedente – vedremo come l'utente alla luce di quanto appena discusso sarà in grado di comprendere il reale funzionamento delle proprie applicazioni e di come esse vengano gestite dal sistema in ogni circostanza, più o meno favorevole.

Capitolo 6 – Hello Word con Eclipse

A questo punto, la filosofia del sistema è stata introdotta, gli strumenti necessari illustrati non resta altro da fare che partire con il primo progetto.

Lo scopo di questo capitolo non è tanto quello di scrivere un vero e proprio programma quanto quello di farci accompagnare dall'IDE nella creazione di un progetto Android per poterne vedere la struttura, innanzitutto, e mandarlo in esecuzione in modo da verificare la corretta preparazione della nostra macchina di sviluppo.

Prendiamo contatto con l'IDE: creazione di un progetto

Probabilmente il programmatore novizio di Android sarà già più che svezzato nel mondo Java quindi non dovrebbe avere problemi ad orientarsi in Eclipse. Comunque, riepilogando, per poter testare la propria macchina di sviluppo è necessario innanzitutto creare un nuovo progetto Android:

- assicuriamoci di avere aperto la Perspective di Eclipse relativa a Java (e non a Java EE né a

nessun altro tipo di framework). Per farlo, andiamo sul menu Window, poi Open Perspective. Se viene visualizzata la voce Java, selezioniamola; altrimenti possiamo procedere con il punto seguente;

- muovendoci nel menu *File* invochiamo *New, Android Application Project;*
- seguiamo la procedura di creazione guidata. La serie di schermate (circa 4 o 5) che si susseguono

ci chiede di inserire una serie di impostazioni. L'unica assolutamente obbligatoria è il **nome dell'applicazione**.

Diciamo che il nome scelto per questa prima applicazione sia – nemmeno a dirlo – *HelloWorld*, **tutte le restanti** impostazioni **al momento possono essere lasciate con i valori di default**, saranno comunque modificabili successivamente. Si può concludere la creazione del progetto selezionando

sempre il pulsante *Next* finché non diventerà attivo quello denominato *Finish*.

L'**architettura di progetto** così impostata è costituita da un certo numero di file e cartelle.

Tutti sono importanti ma gli elementi tra i quali il programmatore dovrà sapersi muovere al più presto con scioltezza sono:

- la **cartella *src*** che conterrà tutto il codice Java che scriveremo;
- la **cartella *res*** in cui

risiederanno le cosiddette risorse dell'applicazione per la maggior parte configurate in XML ma non solo;

- il file ***AndroidManifest.xml*** anch'esso in XML che custodirà configurazioni e ruoli dei componenti della nostra app.

Un progetto creato in questa maniera da Eclipse è funzionante, sebbene non contenga nessuna funzionalità particolare. Al momento, quindi, non modifichiamo nulla e **passiamo subito al suo avvio immediato**.

Esecuzione del progetto: dispositivo emulato o reale?

Per eseguire il test è necessario che si abbia a disposizione un dispositivo Android attivo. Può trattarsi di un dispositivo reale – tipicamente smartphone o tablet collegato via USB – o di un sistema emulato (tecnicamente un AVD, *Android Virtual Device*) mediante gli strumenti messi a disposizione da Android SDK.

Esecuzione su un dispositivo emulato

Partiamo da questo secondo caso. Nel menu *Window* (attenzione, per vederlo è necessario trovarsi nella prospettiva Java e non Debug) sono disponibili due voci importantissime: "**Android SDK Manager**" e "**Android Virtual Device Manager**". Il primo serve a profilare il nostro SDK richiedendo lo scaricamento di versioni di Android per le quali vogliamo sviluppare o strumenti aggiuntivi come l'utilissimo HAXM. Per il momento la configurazione di un SDK come lo troviamo in un pacchetto Eclipse appena scaricato va benissimo.

Il secondo strumento, Android Virtual

Device Manager, è ciò che ci serve per **preparare un emulatore**, seguendo questi passi:

1. cliccare sulla voce *Window*, *Android Virtual Device Manager*;
2. nell'interfaccia che si apre c'è un'area che ospiterà l'elenco degli emulatori che creeremo. Alla sua destra **cliccare il pulsante *New*…;**
3. la nuova finestra che si apre, mostrata in figura, permette di **configurare un dispositivo emulato**

semplicemente assegnandogli un nome e la versione di Android che si vuole che esegua oltre ad una serie di impostazioni ulteriori;

4. tornando alla finestra presentata al precedente punto 2, dovremmo vedere il nostro emulatore elencato nell'area bianca. Non resta che **selezionarlo e avviarlo cliccando il pulsante *Start*…**

Esecuzione su un dispositivo fisico reale

Se si vuole utilizzare un **dispositivo reale via USB** non è richiesto apportare modifiche in Eclipse. Usando Windows sono solitamente necessari dei driver reperiti direttamente dal sistema operativo o scaricati appositamente dal programmatore. Qualora, al contrario, si usasse Linux non è richiesta l'installazione di alcun driver, macchina di sviluppo e Android si interfacceranno direttamente.

Lancio dell'applicazione

Dopo il boot del sistema emulato, potremo lanciare la nostra applicazione che verrà eseguita direttamente sul dispositivo. Ciò può essere fatto in modalità **Run o Debug** utilizzando uno dei mezzi messi a disposizione dall'IDE (voci nel menu *Run*, combinazione di tasti o pulsanti sulla barra degli strumenti).

Il risultato dell'esecuzione è molto semplice.

Consiste nella sola apparizione della stringa "Hello world!". **Non è molto ma certifica il raggiungimento dei nostri obiettivi**: la macchina di sviluppo è pronta per mettersi al lavoro e, seconda

cosa, l'impianto di progetto che abbiamo ora a disposizione è funzionante e può essere usato come base per sperimentare tutto ciò che impareremo.

Capitolo 7 – Hello Word con Android Studio

Abbiamo visto quali sono le due IDE principali per sviluppare su Android. Dopo aver trattato la preparazione di un primo progetto con Eclipse ADT, in questo capitolo, affronteremo la medesima operazione con **Android Studio**, ormai considerato il tool ufficiale di sviluppo. Il primo approccio a questo strumento può talvolta rivelarsi meno amichevole del previsto, soprattutto perchè molti sviluppatori provengono da Eclipse.

Primo progetto

Android Studio deriva da IntelliJ di JetBrains, un IDE per il mondo Java particolarmente sensibile alle necessità dello sviluppatore. Al suo interno, include però Gradle, un ottimo strumento per la build automation, molto flessibile ed erede di tutte le principali caratteristiche di predecessori come Apache Ant e Maven.

Si supporrà che Android Studio sia già installato sulla macchina di sviluppo, in quanto tale procedura è già stata illustrata.

All'avvio, l'IDE mostra una finestra di benvenuto, sulla sinistra della quale troviamo un elenco di progetti aperti di

recente, mentre sulla destra è presente un menu che permette l'avvio del lavoro in varie modalità. Selezioniamo la voce *Start a new Android Studio project*.

La creazione guidata che viene avviata si articola in quattro schermate, ognuna dedicata ad un aspetto:

- la prima schermata consente di specificare nome dell'app, Company Domain (di default costituirà la prima parte del package Java che creeremo), nome completo del package Java e collocazione del

progetto nel file system. Su quest'ultimo aspetto, è bene sottolineare Android Studio non fa uso di un unico workspace, come invece avviene su Eclipse;

- la seconda schermata permette di scegliere il **fattore di forma dell'app**: possiamo scegliere tra smartphone/tablet, TV, dispositivi indossabili (*Wear*) o Google Glass. È evidente quanto questa IDE sia pensata specificamente per il

mondo Android nella sua interezza;

- la terza schermata offre la scelta tra vari **template di applicazioni**. Nel caso, ad esempio, di app per smartphone e tablet, saranno proposte varie alternative: un'app vuota, con fragments, con Google Maps intergrato o con i Google Play Services già a disposizione, ed altro ancora;
- tramite l'ultima schermata possiamo configurare la prima

Activity, definendo il nome della classe e del file di layout.

Al termine di questa procedura ci verrà fornito un semplice progetto, in stile “Hello World”, perfettamente funzionante.

Struttura di un progetto

Anche su Android Studio troveremo tre parti principali del progetto: la cartella con il codice Java, la cartella *res* (contenente risorse per lo più realizzate in XML) ed un file di configurazione denominato *AndroidManifest.xml*.

Per prima cosa, si noti che il progetto è contenuto in una cartella denominata *app*. Questo è il modulo di default. L'IDE, infatti, suddivide un progetto in p i ù **moduli**, ognuno dei quali può svolgere un ruolo diverso (libreria Java, libreria Android, inclusione di un progetto esterno, eccetera…). Il modulo *app* include i file manifest, il codice

Java e le risorse.

Dopo il modulo troviamo la sezione **Gradle Scripts**. Qui ci sono i file di build che userà Gradle per trasformare il nostro progetto in un'app funzionante. In particolare, i file di build sono due: uno per tutto il progetto ed uno per il solo modulo app. Vediamo di seguento quest'ultimo, che è quello tipicamente modificato dal programmatore.

```
apply plugin: 'com.android.application'


android {
    compileSdkVersion 21
    buildToolsVersion "22.0.1"

    defaultConfig {
```

```
        applicationId "it.html.helloworld"
        minSdkVersion 14
        targetSdkVersion 21
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            minifyEnabled false
                                                    proguardFiles
getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
        }
    }
}

dependencies {
```

```
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.1.1'
}
```

La prima riga carica il plugin Gradle per android. Questo permette di avere a disposizione la sezione seguente contenuta nella direttiva `android { ... }`. Al suo interno vengono impostati alcuni fattori che nei progetti Eclipse trovano spazio nel file *AndroidManifest.xml*: minimo SDK cui l'app è destinata (`minSdkVersion`), SDK target (`targetSdkVersion`), versione di sviluppo (`versionCode`) e versione pubblica (`versionName`).

La sezione successiva denominata

dependencies, è molto importante per l'espansione delle funzionalità del progetto.

La riga seguente:

compile fileTree(dir: 'libs', include: ['.jar'])*

include nel build path tutti i file *.jar* che trova nella cartella *libs*, mentre la riga:

compile 'com.android.support:appcompat-v7:22.1.1'

include nel progetto la libreria di supporto Appcompat v7 nella versione 22.1.1. Questo tipo di librerie servono per mettere a disposizione di versioni precedenti di Android funzionalità uscite successivamente.

È interessante notare come sia espresso il nome della dipendenza. Esso è costituito da tre parti, separate dal simbolo `:`. Questo tipo di espressione richiama il formato di Maven: `com.android.support` è l'identificativo del gruppo di sviluppo, `appcompat-v7` il nome del progetto da includere e `22.1.1` è la versione del progetto.

Tale funzionalità, per essere inclusa, deve essere stata prima scaricata tramite Android SDK Manager.

Il medesimo formato delle dipendenze deve essere utilizzato per scaricare ulteriori librerie che il programmatore vorrà sfruttare, disponibili presso repository Maven.

Un aspetto da non dimenticare è che, non appena apportata una modifica al file di build, si deve selezionare il pulsante *Sync Project with Gradle file*.

SDK Manager ed emulatori su Android Studio

Su Android Studio esiste un sottomenu raggiungibile da Tools - Android, in cui è possibile rintracciare i comandi per attivare Android SDK Manager e Android Virtual Device Manager.

Il primo, come accennato nei capitoli precedenti, è il pannello per integrare il proprio SDK con ulteriori funzionalità ed aggiornamenti.

Il secondo permette di creare uno o più emulatori, qualora non si vogliano o non si possano eseguire i propri progetti direttamente su dispositivo reale. Il pannello che lo costituisce presenta, in

basso, il pulsante *Create Virtual Device*..., selezionando il quale verrà avviata la creazione di un emulatore, del quale si potranno scegliere modello e caratteristiche. Terminata la creazione, il nuovo emulatore apparirà nella finestra *Your Virtual Devices* e da lì potrà essere avviato.

Design di layout

Un aspetto significativo di Android Studio consiste nell'anteprima di layout praticamente istantanea. Se si apre un file contenente la stuttura del layout – reperibile, ad esempio, nella cartella res/layout – si vede che il suo contenuto può essere mostrato in modalità Design (visuale, incluso nel display di un dispositivo) oppure Text, che mostra il tipico formato XML.

Il pannello *Design* può essere utile per visualizzare rapidamente le modifiche apportate al sorgente XML, oppure per disegnare in modalità visuale l'interfaccia, trascinando direttamente sul display i widget dalla *Palette*.

Inoltre, il pannello Design presenta alcuni menu a tendina che permettono di modificare le condizioni dell'anteprima in termini di modello, orientamento e versione di Android disponibile.

Capitolo 8 – Activity la “prima pagina” dell'applicazione

Il progetto approntato nel capitolo precedente con l’aiuto dell’IDE può essere ora analizzato nel dettaglio. Lo scopo che ci prefiggiamo è quello di osservare da vicino come è fatta un’**Activity**, il primo dei quattro componenti basilari che troviamo nelle applicazioni Android.

Nel nostro progetto ce n’è una ed è l’interfaccia utente che mostra il messaggio “Hello World”. Nonostante la sua semplicità, mette in luce un aspetto fondamentale. Per creare

un’Activity è necessario fare due cose:

- **estendere la classe Activity**, appartenente al framework Android;
- **registrare l’Activity nell’AndroidManifest.xn** mediante l’uso dell’apposito tag XML <activity>. Tra l’altro, questo dettame vale per tutte le quattro componenti fondamentali di un’applicazione.

L'Activity nel codice Java

Il codice Java che realizza l'Activity risiede nella cartella src, come spiegato in precedenza. Il contenuto di un tipico "Hello world" potrebbe essere questo:

```
public class MainActivity extends Activity
{
    @Override
        protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

}
```

La classe si chiama `MainActivity` ed estende `Activity`. Al suo interno viene implementato l'override del metodo `onCreate`. Per il momento, ci accontentiamo di sapere che questo metodo viene invocato alla creazione dell'Activity. Più avanti scopriremo che si tratta di una tappa fondamentale del ciclo di vita di questo tipo di componenti.

A proposito delle due righe di codice presenti all'interno dell'onCreate:

- `super.onCreate(save` invoca il metodo omonimo della classe

base. Questa operazione è assolutamente obbligatoria;

- `setContentView(R.la` specifica quale sarà il "volto" dell'Activity, il suo **layout**. Al momento la dicitura R.layout.activity_main può apparire alquanto misteriosa ma lo sarà ancora per poco, fino al momento in cui verrà illustrato l'uso delle risorse. **Il suo effetto è quello di imporre come struttura grafica dell'Activity il**

contenuto del file activity_main.xml presente nella cartella res/layout.

L'Activity nel file Manifest

Il file AndroidManifest.xml che configura questa applicazione appare così:

```
<manifest
xmlns:android="http://schemas.android.com/apk/re
    package="esempi.android.helloworld"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
```

```
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action
android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHE
/>
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Il nodo <application> contiene le componenti usate nell'applicazione. In questo caso, c'è un nodo <activity> che con l'attributo android:name specifica il nome della classe Java che incarna l'Activity. Se, come in questo caso, non viene specificato un package è sottintesa l'appartenenza della classe al package riportato nel nodo <manifest>, la *root* del file.

Il costrutto intent-filter all'interno serve ad indicare che questa activity è la **main activity del progetto**, in pratica l'interfaccia che accoglierà l'utente all'ingresso nell'applicazione.

E se volessimo altre activity? È

possibile averne? Certamente.

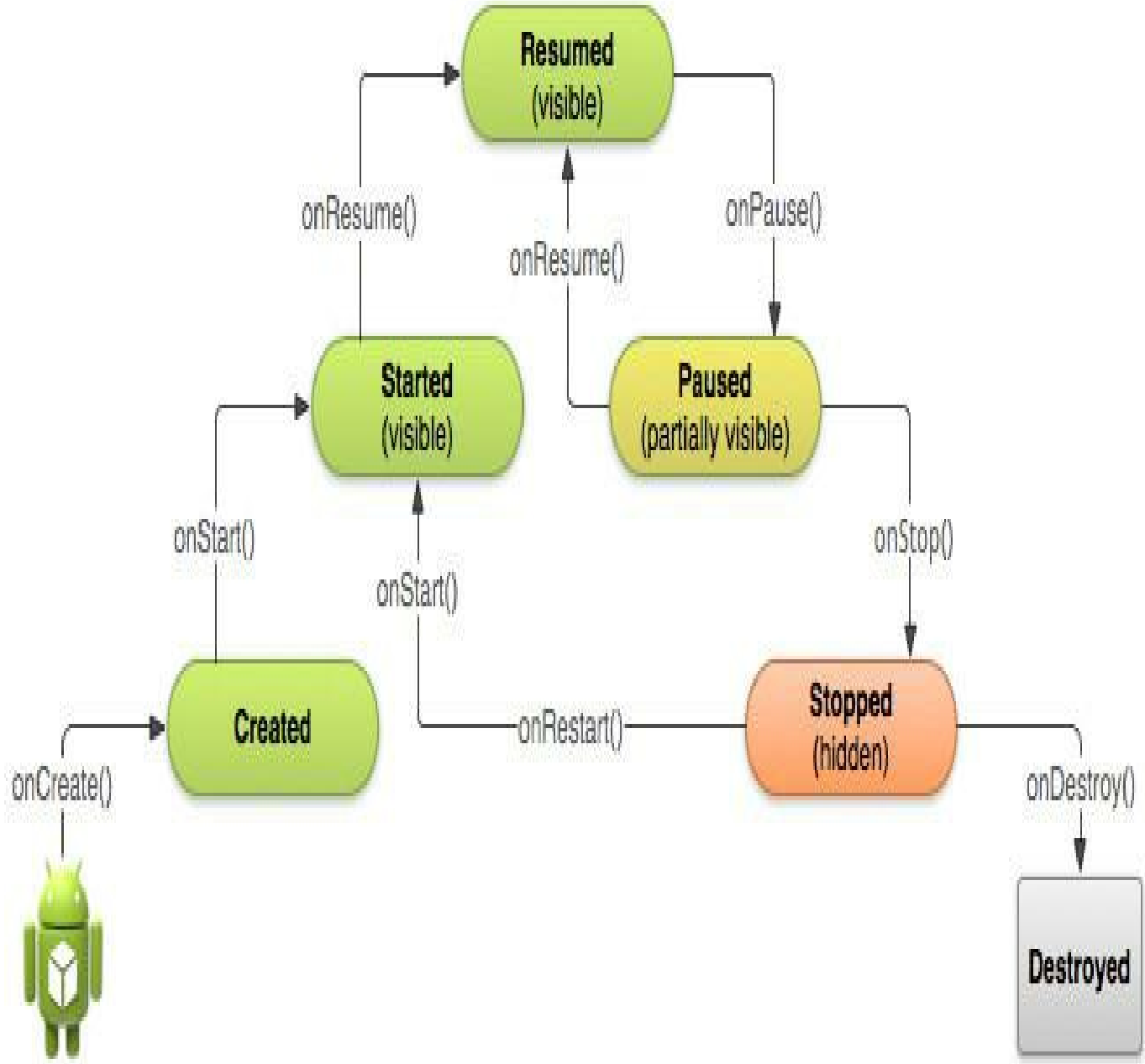
L'importante è che ogni activity venga prodotta seguendo i due passi definiti all'inizio di questa sezione: (1) creare una classe Java che estenda Activity e (2) definire un nodo <activity> in AndroidManifest.xml che riporti nell'attributo android:name il nome della classe corrispondente. A proposito di questo, il sottonodo intent-filter riferito all'action MAIN va indicato solo nell'Activity principale del progetto.

L'utente passerà da un'activity all'altra in maniera simile a come è abituato a fare tra le pagine dei siti Internet ma qui il tutto avverrà mediante il potente meccanismo degli Intent

spiegato a breve.

Capitolo 9 – Il ciclo di vita di un'Activity

Una delle più note illustrazioni della programmazione Android è questa:



La si può trovare sulla documentazione ufficiale, nelle pagine in cui viene spiegato il **ciclo di vita** di un'Activity. Si tratta di una serie di stati

attraverso i quali l'esistenza dell'Activity passa. In particolare, nell'illustrazione riportata, gli stati sono rappresentati dalle figure colorate. L'ingresso o l'uscita da uno di questi stati viene notificato con l'invocazione di un metodo di callback da parte del sistema. Il codice inserito in tali metodi dovrà essere allineato con la finalità del metodo stesso affinchè l'app possa essere "un buon cittadino" dell'ecosistema Android.

Ad esempio, il primo metodo di callback che viene raffigurato è `onCreate()` ed è proprio l'onCreate di cui abbiamo fatto l'override nell'implementazione dell'Activity vista nei capitoli precedenti.

Quando un'activity va in esecuzione per interagire direttamente con l'utente vengono obbligatoriamente invocati tre metodi:

- **onCreate**: l'activity viene creata. Il programmatore deve assegnare le configurazioni di base e definire quale sarà il layout dell'interfaccia;
- **onStart**: l'activity diventa visibile. È il momento in cui si possono attivare funzionalità e servizi che

devono offrire informazioni all'utente;

- **onResume**: l'activity diventa la destinataria di tutti gli input dell'utente.

Android pone a riposo l'activity nel momento in cui l'utente sposta la sua attenzione su un'altra attività del sistema, ad esempio apre un'applicazione diversa, riceve una telefonata o semplicemente – anche nell'ambito della stessa applicazione – viene attivata un'altra Activity. Anche questo percorso, passa per tre metodi di callback:

- **onPause** (l'inverso di onResume) notifica la cessata interazione dell'utente con l'activity;

- **onStop** (contraltare di onStart) segna la fine della visibilità dell'activity;

- **onDestroy** (contrapposto a onCreate) segna la distruzione dell'activity.

Nel seguito della guida verranno offerti casi pratici di utilizzo ma per il momento ci teniamo su una linea

piuttosto teorica. Intanto si consideri che i metodi di callback sono concepiti a coppie (un metodo di avvio con un metodo di arresto: `onCreate-onDestroy`, `onStart-onStop`, `onResume-onPause`) e solitamente il lavoro fatto nel metodo di avvio – in termini di funzionalità attivate e risorse allocate – verrà annullato nel corrispondente metodo di arresto.

La prima situazione che si mostrerà favorevole ad illustrare l'invocazione dei più importanti metodi di callback sarà l'uso degli Intent per passare da un'activity all'altra. Si tratta, in fin dei conti, di una pratica comunissima nella programmazione Android.

Capitolo 10 – Gestire le risorse e gli asset

Nelle applicazioni Android il codice Java richiama spesso degli elementi interni al progetto come file XML, stringhe, numeri, immagini ed altro ancora. Il modo migliore per conservare tutti questi "valori" a disposizione dell'applicazione è collocarli all'interno della cartella di progetto denominata *res* e gestirli mediante l'apposito meccanismo delle **risorse**.

Il capitolo che inizia si occupa proprio di questo: comprendere bene come gestire le risorse di un'applicazione e come utilizzarle richiamandole nel codice. In

conclusione, si vedrà un altro meccanismo di gestione di file all'interno del progetto: gli **assets**.

Dove si trovano le risorse

Se si dà uno sguardo ad un qualsiasi progetto per Android si può vedere che *res* conta diverse sottocartelle i cui nomi non sono affatto casuali.

Tra quelle di più comune utilizzo, troviamo:

- **layout**: conterrà l'architettura grafica dei componenti dell'interfaccia utente. Sono definiti in XML in una maniera simile a come viene usato HTML per strutturare le pagine

web;

- **values**: conterrà stringhe, colori, dimensioni e altre tipologie di valori che potranno essere usate in ulteriori risorse o nel codice Java. Importante notare che questi valori costituiranno il contenuto di appositi tag XML (`<string>`, `<dimen>`, etc.) raggruppati in files dal nome solitamente indicativo: strings.xml, dimens.xml, colors.xml e via dicendo. Tali nomi sono frutto di pura

convenzione ma il programmatore può scegliere liberamente come chiamarli;

- **values**: conterrà stringhe, colori, dimensioni e altre tipologie di valori che potranno essere usate in ulteriori risorse o nel codice Java. Importante notare che questi valori costituiranno il contenuto di appositi tag XML (`<string>`, `<dimen>`, etc.) raggruppati in files dal nome solitamente indicativo: strings.xml,

dimens.xml, colors.xml e via dicendo. Tali nomi sono frutto di pura convenzione ma il programmatore può scegliere liberamente come chiamarli;

Come richiamare le risorse

Le risorse vengono compilate in un formato binario ed indicizzate mediante un ID univoco. Tali ID sono conservati in una classe Java, di nome `R`, autogenerata ad ogni modifica e visibile nella cartella *gen* del progetto Android. Abbiamo già incontrato la classe R nell'Activity esaminata in precedenza.

Il codice:

```
setContentView(R.layout.activit
```

indicava che il layout dell'activity era collocato tra le risorse. In particolare, ogni percorso interno alla classe R rispecchia una collocazione di risorse nelle sottocartelle di *res*.

Mediante i loro identificativi, **le risorse sono accessibili sia da codice Java che da altre risorse definite in XML**:

- in Java: tramite `R.tipo_risorsa.nome_r`
- in XML: `@tipo_risorsa/nome_r`

Ad esempio, la risorsa di tipo stringa e nome appname:

```
<string
android:name="appname">Hello
world!</string>
```

potrà essere recuperata, in Java, mediante `R.string.appname` o dall'interno di altre risorse XML con `@string/appname`.

Adattamento multipiattaforma delle applicazioni

La frammentazione dello scenario hardware nel mondo Android resta uno degli scogli più ardui da superare per il programmatore. In questo le risorse giocano un ruolo molto importante. Osservando un tipico progetto Android si può vedere che tra le cartelle interne a res ne appaiono alcune con nomi "canonici" (*menu*, *values*, *layout*) e altre con nomi "modificati" (*drawable-hdpi*, *drawable-mdpi* ma anche *values-v14*, *values-v11*, etc.).

Questo perchè al nome della cartella

si può accodare un suffisso che rappresenta la configurazione del dispositivo con cui potranno essere richiamate le risorse contenute.

Se, ad esempio, *res/layout* conterrà la struttura grafica delle varie interfacce per una qualsiasi configurazione, *res/layout-land* conterrà layout utilizzabili solo quando il dispositivo è in posizione landscape. Altri modificatori di una certa rilevanza sono quelli che si riferiscono alla lingua del dispositivo: *values-it* saranno le risorse per dispositivi in italiano, *values-en* per quelli in inglese. Di modificatori esiste una collezione grandissima, tutta disponibile sulla documentazione ufficiale.

Un discorso a parte meritano le immagini. I modificatori applicati alle cartelle drawable (*ldpi*, *mdpi*, *hdpi* e via dicendo) sono alcune delle sigle che identificano le **densità dei display**. Questo concetto di densità rappresenta la quantità di pixel per area fisica dello schermo. Programmando per Android è bene abbandonare l'abitudine di misurare in pixel, utilizzando come unità di misura degli elementi grafici i **dp (Density-Indipendent Pixel)**, una specie di pixel "virtuale" indipendente dalla densità del display che permette di mantenere intatte le proporzioni tra gli elementi del layout al variare delle densità.

Gli assets

La documentazione ufficiale elenca tutte le tipologie di risorse che possono essere usate. Esiste anche un tipo di risorsa "grezza" collocabile nella cartella *res/raw*. Vi si potrà collocare tutto ciò che non si riesce ad inquadrare in una particolare tipologia. In alternativa alle risorse raw, si possono definire gli assets. Questi esulano dal meccanismo delle risorse e vanno depositati nell'omonima cartella di progetto. Non vengono né compilati in formato binario né etichettati con un ID univoco. La loro fruizione da parte dell'applicazione avverrà mediante uno stream che potrà essere richiesto ad una classe Java di nome AssetManager.

Capitolo 11 – Intent e messaggi

Quando si è parlato delle componenti che rappresentano i blocchi costitutivi di un'app, si è accennato al ruolo degli Intent. Approfondendo il discorso possiamo dire che rappresentano una forma di messaggistica gestita dal sistema operativo con cui **una componente può richiedere l'esecuzione di un'azione da parte di un'altra componente**.

Sono uno strumento molto duttile anche se gli utilizzi più comuni ricadono in queste tre casistiche:

1. avviare un'Activity;
2. avviare un Service;
3. inviare un messaggio in broadcast che può essere ricevuto da ogni applicazione.

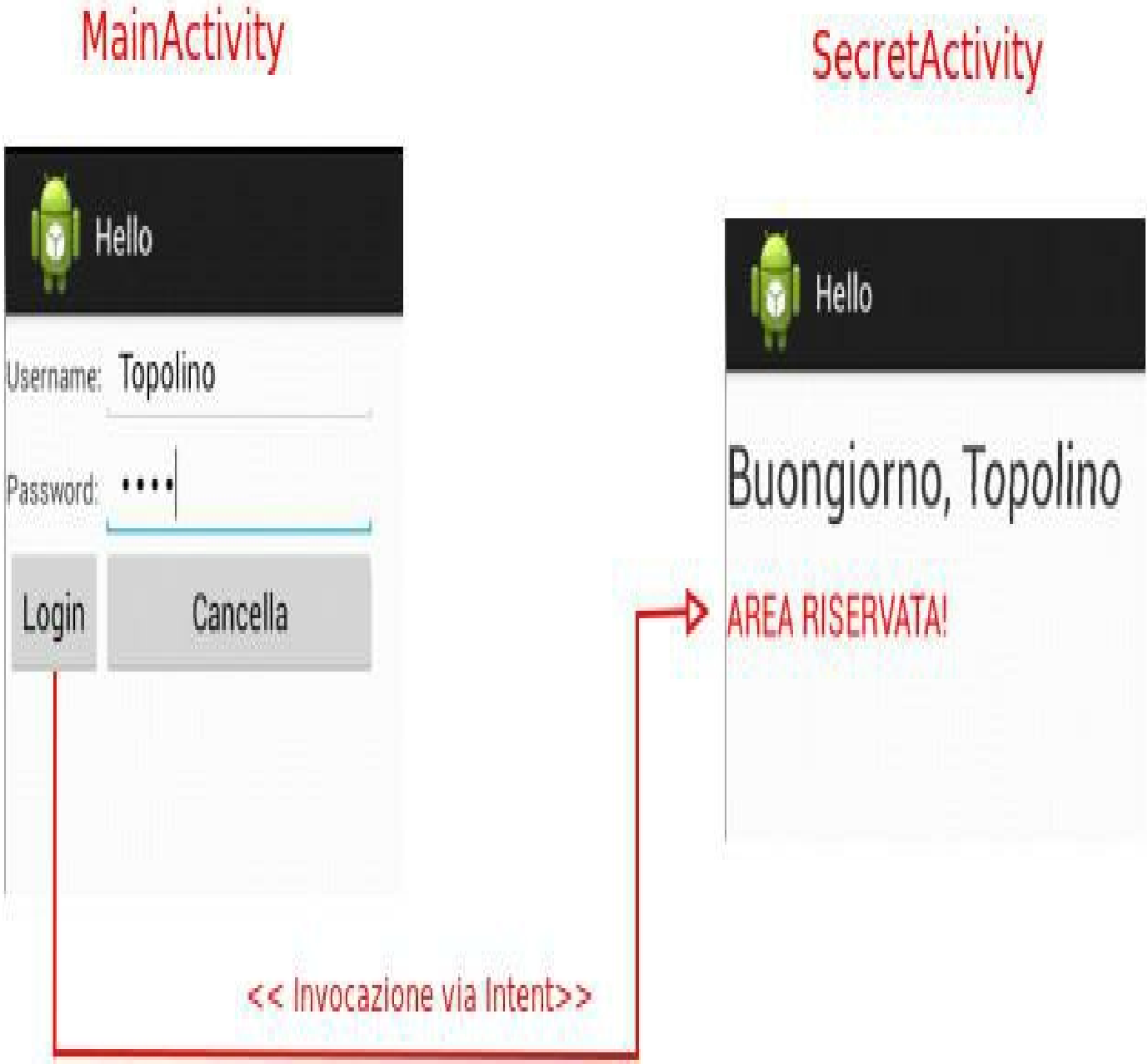
Gli Extras

Un altro aspetto molto utile degli Intent è che essi, nel recapitare questo messaggio, hanno a disposizione una specie di "bagagliaio", in cui custodiscono dati che possono essere letti dal destinatario. Questi valori condivisi mediante Intent vengono generalmente chiamati Extras e possono essere di varie tipologie, sia appartenenti a classi più comuni che ad altre purchè serializzabili. La gestione degli Extras negli Intent funziona in maniera simile ad una struttura dati a mappa: con dei metodi *p u t* viene inserito un valore etichettato con una chiave e con i corrispondenti metodi *get* viene prelevato il valore, richiedendolo

mediante la chiave di riconoscimento.

L’esempio: un form di login

Prendiamo a titolo di esempio il caso più comune, solitamente utilizzato presto dal programmatore Android neofita, l’attivazione di un’Activity da parte di un’altra. Osserviamo quanto appare in figura:



Abbiamo **due activity**:

- **MainActivity** contiene

un semplice form di login. Dopo aver inserito username e password viene controllata la validità dei dati ed in caso positivo viene invocata l'apertura di un'altra activtiy;

- **SecretActivity** è l'area accessibile solo mediante login e contiene – immaginiamo – dati riservati.

All'ingresso della seconda Activity, **l'applicazione vuole dare il benvenuto all'utente** ma per farlo ha bisogno di sapere come si chiama.

Tutto ciò che serve è già incluso nel meccanismo degli Intent.

A livello di codice, nella MainActivity, una volta ottenuto il successo nel login troveremo:

```
Intent i=new Intent(this,SecretActivity.class);
i.putExtra("username", account_username);
startActivity(i);
```

Le tre operazioni rappresentano:

- dichiarazione dell'Intent come normale oggetto Java. In questo caso avremo un cosiddetto **intent**

esplicito in quanto appare chiaramente il nome della classe che verrà invocata;

- viene inserito, tra gli Extras, una String, la variabile `account_username`, che sarà trasportata con l'Intent fino a destinazione ossia la classe SecretActivity. Lo scopo è inserire in questa stringa il nome dell'utente che ha effettuato il login. Notare che l'*extra* viene etichettato con una

chiave, in questo caso “username”. Ciò perchè possono essere trasportati più Extras per ogni Intent e dovrà essere possibile distinguerli per utilizzarli;

- infine il metodo `startActivity` dimostra quale azione vogliamo attivare con questo Intent, si tratta, in questo caso, dell’avvio di un’Activity.

Nel metodo `onCreate` della seconda Activity, quella con il contenuto riservato, troveremo le seguenti righe:

Intent i=getIntent();

String username=i.getStringExtra("username");

Notiamo subito che l'Activity attivata si trova a disposizione, mediante `getIntent()`, l'Intent che ne ha provocato l'attivazione. Lo può utilizzare per recuperare la stringa passata, contenente il nome utente.

A livello di ciclo di vita, che succede?

Il passaggio da un'Activity ad un'altra coinvolge i cicli di vita di entrambe. La prima, quella messa a riposo, dovrà passare almeno per `onPause` (cessazione interazione con l'utente) e `onStop` (activity non più visibile) mentre la seconda percorrerà la catena di creazione `onCreate-onStart-onResume`.

Ma in che ordine avverrà tutto ciò? La priorità del sistema è **il mantenimento della fluidità della *user-experience***. Per questo la *consecutio* delle operazioni sarà:

- la **prima Activity passa per onPause** e viene fermata in stato Paused;
- la **seconda Activity va in Running** venendo attivata completamente. In tale maniera l'utente potrà usarla al più presto non subendo tempi di ritardo;
- a questo punto, mentre l'utente sta già usando la seconda Activity, il sistema può invocare **onStop sulla prima**.

Capitolo 12 – Il layout di un'app Android

Un'Activity ha bisogno di un volto, di un suo aspetto grafico. Sempre. Anche nei casi più semplici, come quando si limita a stampare la stringa "Hello World!".

La struttura grafica di un'Activity prende il nome di **Layout** ed è una delle prime competenze di cui ha bisogno un neo-programmatore Android.

Abbiamo già incontrato i layout nel corso di questa guida. È successo quando si è parlato della prima Activity, ma anche quando si è illustrato l'organizzazione delle risorse. Ora è

arrivato il momento di entrare nel vivo del discorso scoprendone le tipologie più comuni e analizzandole sia in termini di caratteristiche che di finalità.

L’interfaccia grafica

Le interfacce utente in Android possono essere create in modo procedurale o dichiarativo. Nel primo caso si intende l’implementazione dell’interfaccia grafica nel codice: ad esempio, in un’applicazione Swing scriviamo il codice Java per creare e manipolare tutti gli oggetti dell’interfaccia utente, come JButton, JFrame e così via.

L a **creazione dichiarativa** non richiede la scrittura di codice: l’esempio tipico del metodo dichiarativo è rappresentato dalla creazione di una pagina web statica, in cui utilizziamo l’HTML per descrivere

cosa vogliamo vedere nella pagina. L'HTML è dunque dichiarativo.

Android permette la creazione di interfacce sia procedurali sia dichiarative: possiamo creare un'interfaccia utente completamente in **codice Java** (metodo procedurale) oppure possiamo creare l'interfaccia utente attraverso un **descrittore XML** (metodo dichiarativo). Android inoltre permette anche un approccio ibrido, in cui si crea un'interfaccia in modo dichiarativo e la si controlla e specifica in modo procedurale (si richiama il descrittore XML da codice e si continua a lavorare da lì).

Se diamo un'occhiata alla

documentazione Android, vedremo che per i componenti delle interfacce utente gli attributi presenti nelle API Java e quelli utilizzabili negli XML dichiarativi sono gli stessi: questo dimostra che in entrambi i casi abbiamo a disposizione le medesime possibilità. Dunque quale metodo è meglio utilizzare?

Entrambi i metodi sono validi, anche se Google suggerisce di usare la metodologia dichiarativa perché spesso il codice XML è più corto e semplice da leggere e capire rispetto al corrispondente codice Java, ed inoltre è molto meno probabile che cambi nelle versioni future della piattaforma.

Nella prassi, su Android **un layout**

viene progettato in XML, in una modalità che ricorda molto l'uso di HTML per le pagine web. Ciò è particolarmente apprezzato da tutti quei programmatori che provengono da esperienze professionali o percorsi didattici nel settore. Per questi motivi, nel seguito della guida ci affideremo principalmente a questa modalità.

Gli IDE offrono strumenti visuali per disegnare layout con approccio drag-and-drop e visualizzazioni di anteprima molto utili. Nonostante questi strumenti, nel tempo, siano diventati sempre più usabili ed intuitivi, la conoscenza della sintassi XML per le UI e le corrispondenti classi Java restano un fattore imprescindibile.

Tipi di Layout

Nel framework Android sono stati definiti vari tipi di layout ma ce ne sono tre di utilizzo molto comune che permettono di affrontare ogni situazione:

1. **LinearLayout**: contiene un insieme di elementi che distribuisce in maniera sequenziale dall'alto verso il basso (se definito con orientamento verticale) o da sinistra a destra (se ha orientamento orizzontale, il valore di default). È un

layout molto semplice e piuttosto naturale per i display di smartphone e tablet;

2. **TableLayout**: altro layout piuttosto semplice, inquadra gli elementi in una tabella e quindi è particolarmente adatto a mostrare strutture regolari suddivise in righe e colonne come form o griglie. È piuttosto semplice da usare e ricorda molto le tabelle HTML nelle pagine web con i ben noti tag <table> <tr> <td>;

3. **RelativeLayout**: sicuramente il più flessibile e moderno. Adatto a disporre in maniera meno strutturata gli elementi, ricorda un po' il modo di posizionare flottanti nelle pagine web. Essendo "relative" gli elementi si posizionano in relazione l'uno all'altro o rispetto al loro contenitore, permettendo un layout fluido che si adatta bene a display diversi. Rispetto agli altri due è

ricco di attributi XML che servono ad allineare e posizionare gli elementi tra loro.

La figura che segue mostra tre semplici esempi realizzati con layout diversi.

TableLayout
per una struttura ordinata

RelativeLayout
Comodo per posizionare gli elementi in maniera più variegata

LinearLayout
a orientamento verticale. I controlli discendono dall'alto

In generale, non c'è nessun lavoro precluso ad un particolare tipo di layout. Il programmatore imparerà col tempo e la pratica ad associare la struttura grafica che deve realizzare allo strumento più adatto a progettarla.

Capitolo 13 – Definire layout in XML, il markup di base

Dopo aver classificato i principali layout in base a tipologia e finalità, passiamo all'aspetto pratico, il vero e proprio markup XML necessario a definirli.

Elementi comuni nei layout

Prima di passare agli esempi definiamo alcuni elementi che accomunano le sintassi di tutti i layout. Innanzitutto, gli attributi XML utilizzati per la maggior parte proverranno da un namespace avente URI *http://schemas.android.com/apk/res/and* Per questo motivo quando definiremo layout in un progetto Android il nodo *root* che conterrà tutti gli elementi mostrerà al suo interno la dichiarazione

xmlns:android="*http://schemas.android.com/apk/r*

e darà senso al prefisso android: che verrà usato per tutti gli attributi nel file.

Secondo aspetto comune non solo ai layout ma anche a tutti gli elementi in essi contenuti, la presenza obbligatoria di due attributi: **layout_width** e **layout_height**, che definiscono la capacità dell'elemento di estendersi, rispettivamente, in larghezza (width) o altezza (height). Il loro valore può essere una dimensione, espressa in dp, come già spiegato, o una costante da scegliere tra:

- **wrap_content**: l'elemento sarà alto o largo a sufficienza per includere il suo contenuto;

- **match_parent**: l'elemento si estenderà in altezza o in larghezza fino a toccare il suo contenitore.

Quando si andrà ad impostare layout_heigth (o layout_width) l'IDE suggerirà un terzo valore possibile, *fill_parent*. Questo rappresenta un sinonimo di match_parent ma non va usato in quanto ormai deprecato.

Nel prosieguo di questo capitolo, verrà presentata la sintassi di base dei principali layout. Gli elementi posizionati all'interno dei layout potranno essere altri layout annidati o widget, termine con cui si indicano tutti i

controlli per interfacce utente. Nell'ultimo esempio faranno la loro comparsa TextView e Button, molto comuni nelle UI Android. Una spiegazione più dettagliata dei widget sarà presentata nei prossimi capitoli.

Sintassi dei layout

Il **LinearLayout** riceve con l'attributo **orientation** la sua connotazione principale. Con esso si dichiara in quale senso verranno disposti gli elementi, orizzontalmente (il default) o verticalmente.

Un esempio:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/re
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
        . . .
        . . .
</LinearLayout>
```

Il **TableLayout** viene specificato mediante due tag: TableLayout e TableRow.

```
<TableLayout
xmlns:android="http://schemas.android.com/apk/re
    android:layout_width="wrap_content"
    android:layout_height="match_parent">
      <TableRow
        android:layout_width="wrap_content"
       android:layout_height="match_parent"
          . . .
          . . .
      </TableRow>
          . . .
          . . .
</TableLayout>
```

TableLayout rappresenta la tabella

nel suo complesso mentre ogni nodo **TableRow contiene tutti gli elementi di una riga**. Il concetto di colonna viene reso in automatico, ogni elemento in un TableRow costituisce una colonna.

Il **RelativeLayout** sfrutta gli attributi per definire posizionamenti. Sono molti ma piuttosto intuitivi.

A scopo di orientamento, segue una tabella riassuntiva che ne raggruppa le diverse categorie in base alla finalità. Il tipo di valore assegnato a questi attributi può essere booleano (true o false) o l'id di un elemento appartenente al layout.

layout_alignParentTop layout_alignParentBottom layout_alignParentLeft layout_alignParentRight	**Allineamento con il contenitore**: attributi che definiscono se l'elemento deve allinearsi ad uno dei bordi del proprio contenitore. Il valore di questo attributo è di tipo *booleano*
layout_alignTop layout_alignBottom layout_alignLeft layout_alignRight	**Allineamento con altro elemento**: attributi che definiscono se l'elemento deve allinearsi ad uno dei bordi di un altro elemento del layout. Il valore di questo attributo sarà l'*id dell'elemento* con cui allinearsi
layout_above layout_below layout_toLeftOf layout_toRightOf	**Posizionamento relativo ad un altro elemento**: indicano se l'elemento si trova, rispettivamente, sopra, sotto, a sinistra o a destra del componente il cui *id* è il valore dell'attributo

Il frammento di XML che segue mostra un esempio di RelativeLayout con una TextView collocata in alto a sinistra ed un Button in basso al centro:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/re
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:layout_alignParentTop="true"
          android:layout_alignParentLeft="true"
                android:text="Esempio di Relative
Layout"/>
        <Button
          android:layout_width="90dp"
```

```
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        android:text="Clicca qui!"/>
</RelativeLayout>
```

Capitolo 14 – View: le basi dell'interfaccia grafica

Una classe che incontreremo spesso nelle interfacce utente Android è **View**. Con questo termine intendiamo un qualunque elemento che appare in un'interfaccia utente e che svolge **due funzionalità**:

- mostra un aspetto grafico;
- gestisce eventi relativi all'interazione con l'utente.

Una classe derivata da View è

ViewGroup. Esso è, al contempo, un tipo di View e un contenitore di altre View. Tanto per fare un esempio di ViewGroup pensiamo ai layout, sono tipici raggruppatori di View.

Classificazione di View

Tutto ciò che tratteremo come un controllo utente in Android sarà direttamente o indirettametne discendente di una View.

Nel corso della guida, le tipologie di View da presentare si articoleranno per lo più in tre categorie:

- i **layout** di cui si è discusso nei precedenti capitoli;
- i **widget** che in Android rappresentano i controlli form sono delle View. Il loro utilizzo,

spiegato a breve, si snoderà tra posizionamento nei layout e configurazione dei loro gestori di evento;

- gli **AdapterView** sono delle View, generalmente nello stesso package dei widget, che collaborano nella realizzazione del *pattern Adapter*. Gli AdapterView e gli Adapter saranno illustrati diffusamente nei prossimi capitoli e rappresentano nel percorso di apprendimento un

argomento centrale della UI in Android.

View e id

A livello di ereditarietà, View contiene in sé le caratteristiche comuni a tutti gli elementi che popolano le interfacce Android.

Prima di tutto gli id. Degli id ne abbiamo parlato quando si sono spiegate le risorse e la classe R. Sono riapparsi parlando degli attributi del RelativeLayout. In generale può essere necessario etichettare un elemento della UI per potervi fare riferimento nel codice Java o in altre risorse XML.

Nei file di risorse, gli attributi *id* hanno un valore definito come @+id/*identificatore* dove per identificatore si intende il nome dell'id

scelto dall'utente. Il simbolo + apposto dopo la @ indica che se l'id con quel nome non è stato ancora definito nel sistema sarà definito per l'occasione.

Se troveremo un elemento TextView così configurato:

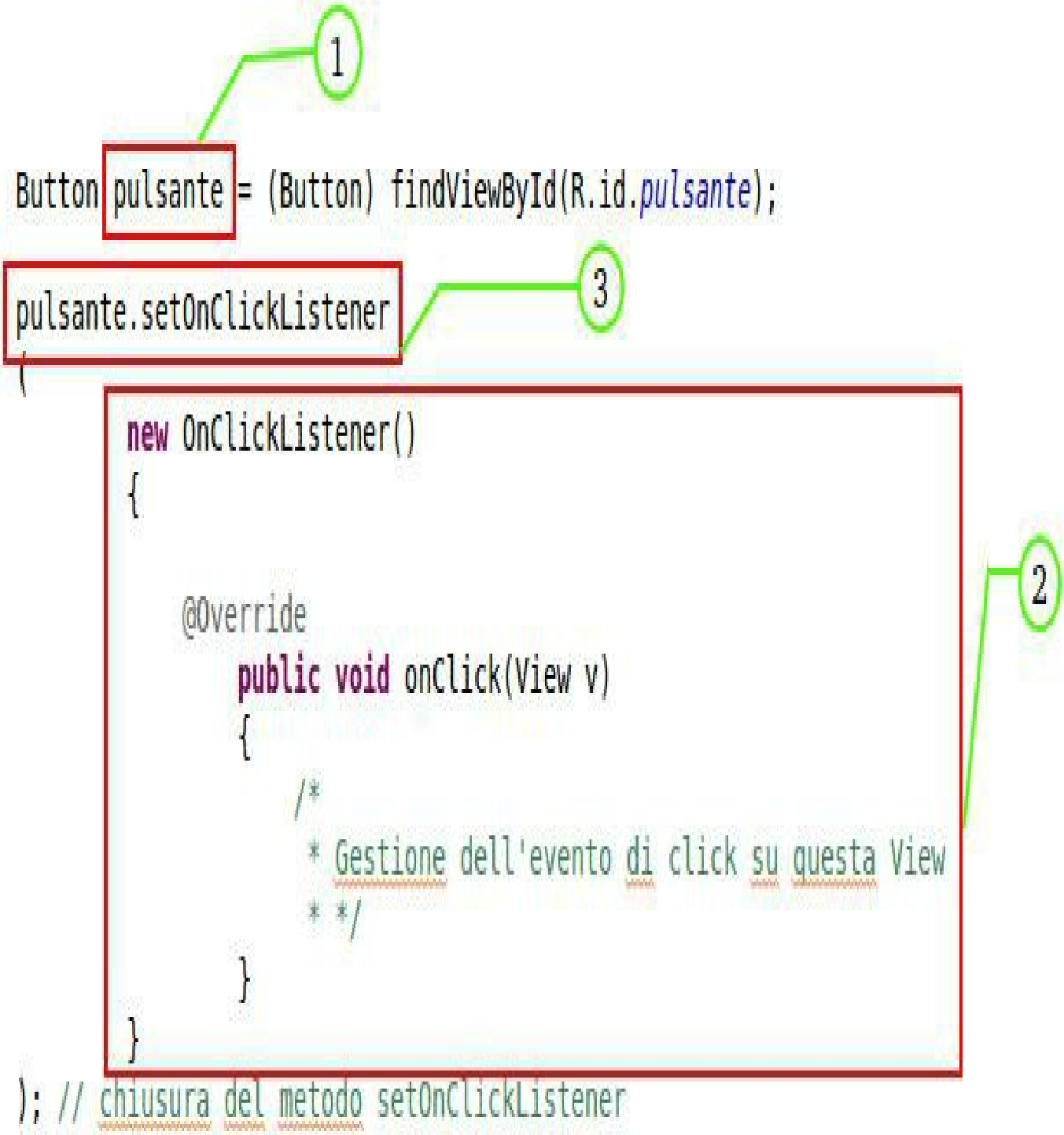
```
<TextView
        android:id="@+id/nome"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        . . .
        . . .
/>
```

significherà che gli si potrà fare riferimento usando il suo id:

- in XML con @id/nome, ad esempio in un attributo del RelativeLayout;
- nel codice Java come R.id.nome.

La gestione degli eventi viene realizzata con il meccanismo dei *listener*.

Nell'immagine che segue, uno stralcio di codice Java mostra una View – in questo caso un Button – che si prepara a gestire un evento di click.



È un esempio che rappresenta i tratti salienti di qualunque gestione degli eventi, i numeri raffigurati indicano:

1. il recupero di un riferimento alla View della quale verranno monitorati gli eventi;
2. la definizione di un oggetto (anonimo in questo caso) che contiene un metodo onClick definito appositamente per gestire l'evento di click;
3. la registrazione del listener tramite un metodo *setter* affinché il pulsante sappia chi è l'oggetto a cui delegare

la gestione dei click.

Nonostante l'esempio sia specifico per un tipo di evento, il meccanismo appartiene a tutte le View e potrà essere riprodotto per qualunque evento. I tre punti cardine rimarranno sempre gli stessi.

Oltre al click le View Android sono in grado di gestire eventi di qualsiasi genere:

- cambiamento del focus;
- pressione di una chiave hardware;
- evento di LongClick, il

click lungo;

- gestione del touch;
- molti altri ancora….

Capitolo 15 – Widget: I componenti interattivi

In questo capitolo faremo conoscenza da vicino con i widget. Che siano tutti discendenti della classe View ormai l'abbiamo imparato. Ma quali sono i widget più comuni in Android? Ne elenchiamo alcuni:

- **TextView**: la classica label. Serve a rappresentare del testo fisso;
- **EditText**: corrisponde all'inputbox di altre tecnologie. Viene usata

per permettere l'inserimento del testo. Mediante attributi XML può essere adattata alle proprie necessità. Molto utile è android:inputType i cui valori definiscono i formati più comuni di input (date, password, testo, etc.);

- **Button**: è il pulsante. La casistica più comune comporterà la gestione dell'evento click per attivare una qualche reazione nell'Activity;
- **CheckBox**: come ci si

aspetta definisce un classico flag che può essere attivato o disattivato. Nel codice Java leggeremo il suo stato (checked o unchecked) mediante un valore booleano;

- **Radio**: esistono i radiobutton come in ogni altra tecnologia per interfacce utente. Vengono solitamente usati per definire valori alternativi tra loro come il sesso di una persona (maschio/femmina), possesso di facoltà

(automunito Sì/No), etc.

L’elenco dei widget sarebbe sconfinato. Ne esistono veramente di ogni tipologia per poter aiutare l’utente ad introdurre gli input più variegati. Per ulteriori approfondimenti, fare pure affidamento alla documentazione ufficiale.

Widget al lavoro: un esempio

Nel capitolo relativo agli Intent, si era immaginato un form di login

Vediamo come potrebbe essere nel dettaglio.

<TableLayout
xmlns:android="http://schemas.android.com/apk/re

```
android:layout_width="match_parent"
android:layout_height="match_parent">
<TableRow
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
     <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Username:"/>
     <EditText
    android:layout_width="@dimen/width"
    android:layout_height="wrap_content"
    android:inputType="text"
    android:id="@+id/username" />
</TableRow>
<TableRow
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Password:"/>
        <EditText
        android:layout_height="wrap_content"
        android:inputType="textPassword"
        android:id="@+id/password" />
</TableRow>
<TableRow
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <Button
        android:layout_width = "wrap_content"
```

```
        android:layout_height = "wrap_content"
        android:onClick="login"
        android:text="Login"/>
        <Button
        android:layout_width = "wrap_content"
        android:layout_height = "wrap_content"
        android:onClick="cancel"
        android:text="Cancella"/>
    </TableRow>
</TableLayout>
```

Notiamo gli aspetti salienti:

- come layout è stato usato un **TableLayout** particolarmente adatto

alla realizzazione di form che spesso mostrano una struttura a griglia molto regolare;

- tra i controlli usati alcuni mostrano l'attributo **android:id**. Serve per lo più alle EditText usate per inserire username e password dell'utente che sta tentando il login. L'id sarà indispensabile per recuperare dal codice Java i valori inseriti;
- i due widget Button presentano l'attributo **onClick** usato per

definire quale metodo dell'Activity si occuperà della gestione del click sul pulsante. Rappresenta una scorciatoia rispetto all'uso dei listener.

Il primo Button presenta questa valorizzazione di onClick:

android:onClick="login"

Ciò comporta che nell'Activity dovremo trovare un metodo della seguente forma:

public void login(View arg0)

dove il nome del metodo corrisponde al valore dell'attributo onClick e il parametro View in input rappresenta il

widget che ha sollevato l'evento, in questo caso il Button.

Uno sguardo da vicino al metodo login nell'Activity ci mostra come i widget vengono richiamati nel codice Java.

```
public void login(View v)
{
        EditText username=(EditText) findViewById(R.id.username);
        EditText password=(EditText) findViewById(R.id.password);
        String account_username=username.getText().toString();
        String account_password=accounts.get(username.getText(
        . . .
        . . .
```

```
}
```

Il metodo findViewById viene usato per recuperare il controllo corrispondente all'id.

Anche in questo caso R.id.username corrisponde all'attributo android:id="@+id/username" che è stato assegnato all'EditText. Dando uno sguardo alla documentazione dei widget, si dovrà di volta in volta riconoscere quei metodi che servono a recuperare il valore del controllo.

In elementi come TextView e EditText rivolti al trattamento del testo, con un metodo getText() si potrà leggere il contenuto e gestirlo come una stringa.

Capitolo 16 – Creare un menu

Le applicazioni per Android fanno largo uso di menu per offrire un'interazione con l'utente più vicina a quella dei tradizionali programmi per desktop.

Le tipologie di menu più comuni in Android sono due:

- Options **menu**: è il menu principale dell'applicazione e contiene voci riguardanti operazioni di interesse generale nella vita

dell'app. Pensando ai programmi per desktop può essere paragonato al menu principale contenuto nella barra del titolo;

- **Context Menu**: è un menu invocabile su un singolo componente dell'interfaccia utente. Le voci richiamabili serviranno ad avviare operazioni sull'elemento su cui è stato richiesto il menu. Normalmente un menu contestuale viene attivato con un click lungo su un componente

del layout. Ha le stesse finalità del menu che nei programmi per desktop viene richiamato con il classico “click” sul pulsante destro del mouse.

Definire la struttura del menu

Il primo passo per aggiungere un menu di qualsiasi tipo alla nostra Activity è crearne la struttura. In proposito, va sempre tenuto a mente che **i menu sono risorse**. Quindi il loro layout va definito nella sottocartella *res/menu*. Questo è il punto di partenza della creazione di un menu.

Il seguente codice mostra un layout di menu:

```
<menu
xmlns:android="http://schemas.android.com/apk/re
>

    <item

        android:id="@+id/MENU_1"
```

```
        android:title="Nuova nota"/>
         <item
        android:id="@+id/MENU_2"
        android:title="Elenco note"/>
    </menu
```

Assumiamo che il nome del file sia main.xml. Come si può vedere la sintassi necessaria non è molto articolata. Per poter creare un menu minimale, sono sufficienti due tag: `<menu>` che definisce il menu nel suo complesso e `<item>` che dichiara la singola voce del menu. Gli attributi impiegati nella configurazione sono due:

1. *id* che, come vedremo, saranno molto importanti nella gestione delle voci del menu ;
2. *title* che contiene una stringa che dà il titolo alla voce di menu.

Il risultato è visibile in figura:

Il menu apparirà cliccando l’immagine cerchiata in rosso apposta sulla barra dell’activity.

Attivare il menu nell'activity

Affinchè il menu venga collegato all'Activity è necessario fare override di un metodo denominato `onCreateOptionsMenu`. Quella che segue è l'implementazione utilizzata nell'esempio:

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    MenuInflater inflater=getMenuInflater();
    inflater.inflate(R.menu.main,menu);
    return true;
}
```

Questo metodo, richiesto nell'Activity che desidera il menu, prende come argomento un riferimento ad un oggetto Menu che non dovremo mai preoccuparci di istanziare in quanto sarà già preparato dal sistema. Ciò che resta da fare è configurarlo assegnandogli il layout che abbiamo predisposto nelle risorse. Questo sarà compito delle tre righe:

1. `MenuInflater inflater=getMenuInflater()`: recupera un riferimento ad un inflater di Menu ossia un servizio del sistema in grado di

modellare la struttura dell'oggetto Menu in base alle direttive impostate in XML;

2. `inflater.inflate(R.menu.ma` questo è il momento in cui l'azione dell'*inflating* viene veramente svolta. Il metodo inflate richiede due parametri: la risorsa contenente il layout del del menu e l'oggetto Menu da configurare;

3. `return true`: solo se il valore booleano restituito da onCreateOptionsMenu

sarà true il menu sarà attivo.

Da ricordare che `onCreateOptionsMenu` **verrà invocato una sola volta**, al momento della creazione del menu, cosa che avverrà contestualmente alla creazione dell'activity.

Gestire le voci del menu

Per poter usare il menu manca solo la gestione del click. Questo viene fatto mediante il metodo `onOptionsItemSelected`.

```
@Override
public boolean onOptionsItemSelected(MenuItem item)
    {
        int id=item.getItemId();
        switch(id)
        {
        case R.id.MENU_1:
            /*
                Codice di gestione della voce MENU_1
            */
            break;
```

```
            case R.id.MENU_2:
                /*
                    Codice di gestione della
voce MENU_2
                */
            }
            return false;
        }
```

Come si può vedere nello stralcio di codice, il parametro in input nel metodo è di classe MenuItem e rappresenta la singola voce selezionata. La prima cosa da fare è recuperare l'id della voce, così come è stato impostato nel menu ed in base al suo valore attivare la gestione corretta.

Creare un Context Menu

Finora la trattazione ha riguardato esclusivamente i menu Options. O almeno così sembra. In realtà **i concetti finora espressi vengono applicati anche ai menu contestuali**. Infatti un Context Menu viene creato in maniera del tutto simile ad un menu Options.

Le operazioni da effettuare sono le seguenti:

- definizione della struttura del menu contestuale in un apposito file della cartella res/menu;

- predisposizione del metodo `onCreateContextMenu` in cui verranno eseguite le medesime operazioni svolte nell'`onCreateOptionsMenu`;
- definizione delle azioni di risposta al click sulle voci mediante `onContextItemSelected`;
- effettuazione di **un'operazione molto importante che non è, viceversa, usata negli OptionsMenu**: la registrazione dell'elemento che vuole

usare il menu contestuale.

Visto che il menu contestuale viene richiamato con click lungo su un elemento del layout, si deve segnalare all'activity quale elemento sarà dotato di questa caratteristica. Per fare ciò si invoca il metodo `registerForContextMenu(View v)`, solitamente nell'onCreate dell'activity, e la View che viene passata come parametro di ingresso è proprio il riferimento all'elemento sul quale può essere attivato il menu contestuale.

Capitolo 17 – ActionBar

Iniziando a sperimentare esempi di codice, uno degli aspetti maggiormente evidenti dal punto di vista grafico, è la spessa fascia scura che si trova nella parte superiore dell'applicazione. Il suo nome è ActionBar e la sua introduzione ha rappresentato un elemento fortemente innovativo a partire dalla versione 3 (HoneyComb) di Android.

Non si tratta solo di un "bordo". In realtà, l'ActionBar può essere definita una **"cornice" programmabile** destinata ad ospitare opzioni di navigazione e di interazione di utilità più o meno comune all'intera applicazione, tra cui:

- *actions* ossia pulsanti cliccabili per attivare azioni. Altro non sono in realtà che voci dell'Options Menu collocate sull'ActionBar;

- navigazione con "tab";
- navigazione con menu a tendina;
- campi di ricerca;
- molto altro ancora.

In questo capitolo, si prenderà confidenza con l'ActionBar iniziando a sperimentarne funzionalità utili e già integrabili con quanto si è appreso

sinora. Come di consueto, la documentazione ufficiale offrirà quanto necessario ad ulteriori approfondimenti.

Avere l'ActionBar disponibile

Essendo entrata a regime in Android 3, l'ActionBar è un elemento un po' di confine.

Se si sta programmando per API di livello 11 o superiori quindi per Android 4.x.x (nell'*AndroidManifest.xml* i valori degli attributi *targetSdkVersion* o *minSdkVersion* dovranno essere impostati almeno a 11) l'ActionBar sarà sempre disponibile purchè si abbia un tema "olografico", quindi nel file manifest l'attributo android:theme dovrà essere impostato a Theme.Holo o un suo discendente.

Per avere l'ActionBar in applicazioni destinate anche a versioni di **Android con API minori di 11** (quindi anche Android 2.x.x) si dovrà collegare il proprio progetto alla libreria di supporto appcompat v7 ed inoltre:

- l'Activity dovrà estendere ActionBarActivity;
- il tema dell'Activity dovrà essere Theme.AppCompat o un derivato .

Comandi nell'ActionBar

L'immagine (fonte: documentazione ufficiale Android) mostra una tipica ActionBar popolata con gli elementi più comuni:

1. **icona** dell'applicazione e titolo;
2. due ***action*s**;
3. *action overflo* che ospita altre actions che non hanno trovato posto sull'ActionBar.


1
6:52
Google Play
2
3
MOVIES
TV SHOWS

Icona dell'applicazione e titolo sono configurabili già dal manifest. Aprendo il file AndroidManifest.xml vediamo che il nodo `<application>` ha un attributo android:icon e android:label:

```
<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
        ...
        ...
```

Servono proprio a definire l'icona ed il titolo per l'applicazione. Questi attributi sono presenti anche nei nodi

`<activity>` che permettono pertanto di adottare un'icona e un titolo per la singola Activity.

Le actions, come già accennato, non sono altro che i comandi che abbiamo imparato a gestire nel capitolo riguardante i menu. Infatti nella visione più moderna della programmazione Android, l'ActionBar tende ad assorbire parte del ruolo degli Options Menu dando la possibilità di ospitarne le voci.

Per farlo si dovrà solo mettere mano al layout del menu (reperibile nella cartella di risorse res/menu) impostando opportunamente il valore dell'attributo `showAsAction`, tra i seguenti:

ifRoom	mostra le icone sull'ActionBar compatibilmente con lo spazio disponibile
never	non mostra le voci del menu sull'ActionBar ma solo nell'Options Menu
always	il layout viene forzato a mostrare le voci in ActionBar. E' un Adjust table row e sconsigliabile, meglio optare per ifRoom
withText	Oltre all'icona viene mostrato in ActionBar anche il testo, solitamente collegato all'attributo android:title

Se si ha necessità di combinare più valori per showAsAction lo si può fare sfruttando un `OR` (|), ad esempio `ifRoom|withText`.

Lo stralcio di XML che segue mostra un OptionsMenu che collocherà

entrambe le sue voci sull'ActionBar.

```
<menu
xmlns:android="http://schemas.android.com/apk/re
>
    <item
        android:id="@+id/MENU_1"
        android:showAsAction="ifRoom"
        android:title="Nuovo"
        android:icon="@android:drawable/ic_mer
         <item
        android:id="@+id/MENU_2"
        android:showAsAction="ifRoom"
        android:title="Elenco"
         android:icon="@android:drawable/ic_me
</menu>
```

Eventuali voci del menu che non

troveranno posto nell'ActionBar verranno ospitate nell'action overflow.

Navigazione “all’indietro”

Altra funzionalità che può essere introdotta da subito nella configurazione dell’ActionBar è il supporto alla navigazione all’indietro. L’immagine mostra, cerchiata in rosso, una piccola freccia verso sinistra accanto all’icona dell’applicazione.

Questo elementino di interazione dovrebbe prendere il posto del pulsante hardware comunemente chiamato “Back” o “Indietro”.

Questa tecnica verrà per lo più usata nelle Activity secondarie per tornare a

quella precedente. Per impostarla sarà necessario seguire due passi:

- nell'*AndroidManifest.x* esattamente nel nodo `<activity>` dell'Activity che dovrà mostrare la frecciolina, si specificherà la parent Activity ossia l'Activity a cui si dovrà tornare:

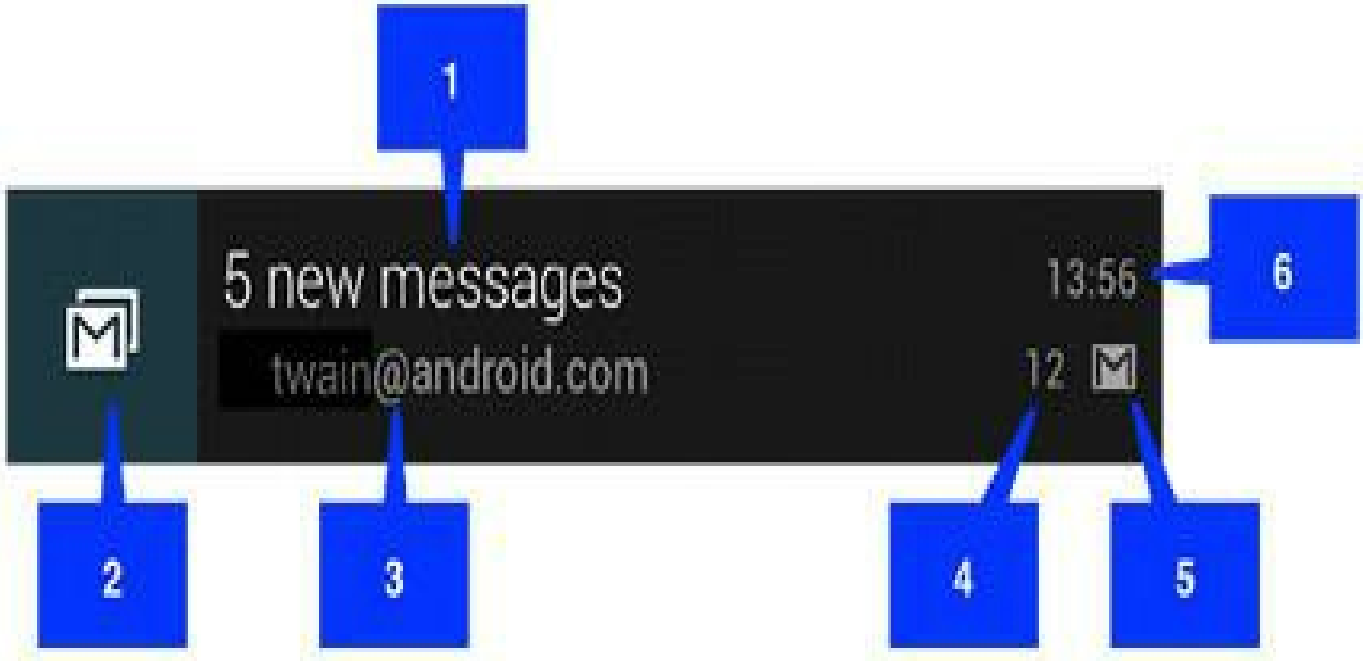
```
<activity android:name=".SecondActivity"
  android:label="Seconda Activity"
  android:parentActivityName=".MainActivity"/>
```

- nell'`onCreate` dell'Activity che mostra la frecciolina, si inserirà la seguente riga di codice

getActionBar().setDisplayHomeAsUpEnabled(tru

Capitolo 18 – Le notifiche di Android

Gli utenti Android sono abituati a ricevere notifiche. Si manifestano con una piccola icona che appare nella cosiddetta “Notification Area” e se ne può consultare il contenuto aprendo il “Notification drawer”, una zona “a scomparsa” sul display. Oltre alla semplicità comunicativa che le contraddistingue e alla familiarità dell’utente con questo meccanismo, vale la pena trattarle in un corso di questo tipo perché offrono un esempio di segnalazione che esula dall’interfaccia dell’applicazione.



Se si osserva la figura (dalla documentazione ufficiale Android) si possono riconoscere i vari elementi che costituiscono una comune notifica. Facciamoci guidare dai numeri indicati:

1. titolo della notifica (*content title*);
2. icona grande (*large*

icon);

3. contenuto della notifica (*content text*);
4. informazioni accessorie (*content info*);
5. icona piccola (*small icon*) che di norma appare anche nella barra del display;
6. ora della notifica (*when*) impostata dal programmatore o di default dal sistema

La prima notifica

Visto che le notifiche appaiono in zone del display non gestite dall'applicazione, dovremo interagire con il sistema mediante un apposito servizio: il **NotificationManager**. Ne recuperiamo un riferimento:

```
NotificationManager notificationManager =
(NotificationManager)
        getSystemService(NOTIFICATION_SERVIC
```

Nonostante la molteplicità di aspetti che contraddistinguono una notifica, ve ne sono **tre assolutamente obbligatori**:

- l'icona piccola;
- il titolo;

- il contenuto.

Questi saranno configurati, rispettivamente, con i metodi `setSmallIcon`, `setTitle` e `setContentText`.

Vediamo subito un esempio:

```
NotificationCompat.Builder n = new NotificationCompat.Builder(this)
.setContentTitle("Arrivato nuovo messaggio!!")
.setContentText("Autore: Nicola Rossi")
.setSmallIcon(android.R.drawable.ic_dialog_em

NotificationManager notificationManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
notificationManager.notify(0, n.build());
```

La creazione della notifica ha seguito due fasi:

- è stata costruita mediante il *Builder* incluso nella classe NotificationCompat. I vari metodi *setter* permetteranno di configurarne i molti aspetti. In questo caso, si è provveduto al minimo indispensabile;
- finita la fase di build, la notifica verrà pubblicata dal NotificationManager

mediante invocazione del metodo `notify`.

La notifica ottenuta nell'emulatore – con il *Notification drawer* aperto – è mostrata in figura:

Collegare un'azione alla notifica

Far apparire notifiche non soddisfa le necessità dell'utente però. Egli è abituato a cliccarvi sopra per ottenere una reazione da parte dell'applicazione. Noi, proseguendo l'esempio, faremo in modo che **il click sulla notifica provochi l'apertura di un'altra Activity** denominata *MessageActivity*.

La preparazione dell'Activity non verrà ripetuta in questa sede considerato lo spazio dedicatole nei capitoli precedenti. Ricordiamo comunque che si dovranno seguire due step fondamentali: creare la classe `MessageActivity` estendendo `Activity` oltre all'eventuale

layout ed inserire un nodo di configurazione del nuovo componente nell'*AndroidManifest.xml*.

L'apertura dell'Activity avverrà mediante `Intent` ma non sarà attivata subito con il metodo `startActivity` bensì sarà predisposta per "usi futuri" mediante la classe `PendingIntent`. Si tratta di una classe che, per così dire, conserva l'Intent e la descrizione dell'azione che esso porta con sé per poterlo attivare successivamente. Ciò che faremo sarà:

- predisporre un normale `Intent` per l'apertura della

`MessageActivity`;

- creazione del `PendingIntent` sfruttando l'`Intent` del punto precedente;
- assegnazione del `PendingIntent` alla notifica mediante il metodo `setContentIntent` del `NotificationCompat.Bu`

Di seguito le modifiche da apportare al codice precedente:

Intent i=new Intent(this,MessageActivity.class);

PendingIntent pi=PendingIntent.getActivity(this, 0, i, 0);

```
NotificationCompat.Builder n = new NotificationCompat.Builder(this)

. . .

. . .

.setContentIntent(pi)

.setAutoCancel(true);


NotificationManager notificationManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);

                    notificationManager.notify(0, n.build());
```

Ora, dopo l'apertura del "Notification drawer", si potrà cliccare sulla notifica e ciò comporterà l'esecuzione dell'azione contenuta nel

`PendingIntent` con conseguente avvio della `MessageActivity`. La notifica scomparirà dalla barra dell'applicazione non appena selezionata, merito dell'invocazione al metodo `setAutoCancel()`.

Notifiche con avviso sonoro

Le notifiche che appaiono sui dispositivi spesso attirano la nostra attenzione con segnalazioni sonore. Per aggiungere questa ulteriore funzionalità all'esempio, dobbiamo recuperare l'Uri del suono che desideriamo:

```
Uri sound = RingtoneManager.getDefaultUri(RingtoneManager
```

e collegarlo alla notifica in costruzione:

```
NotificationCompat.Builder n = new NotificationCompat.Builder(this)

. . .

.setSound(sound);
```

Da questo momento la notifica sarà anche sonora.

Capitolo 19 – Notifiche: Toast e Dialog

Una delle fasi più comuni dell'interazione app-utente è la notifica di messaggi mediante le cosiddette finestre di dialogo. Tra l'altro, la presenza costante delle AlertBox, MessageBox e dialog nelle diverse tecnologie web e desktop di ogni epoca ha reso questa forma di comunicazione particolarmente familiare all'utente.

Toast

Utilizzando un dispositivo Android, una tipologia di notifica che si incontra presto è il cosiddetto Toast. Si tratta di una piccola forma rettangolare nera che appare nella parte bassa del display contenente un messaggio con il testo bianco. La sua visibilità dura poco, qualche secondo, e le sue apparizioni improvvise dal basso gli hanno donato questo nome che richiama letteralmente il modo in cui il pane salta fuori dai tostapane.

Il Toast è la forma di notifica più immediata che esiste e realizzarlo è molto semplice:

Toast.makeText(this, "Ciao a tutti!",

Toast.LENGTH_SHORT).show();

Si fa uso della classe omonima che espone un metodo statico makeText che prepara il messaggio. I tre parametri richiesti sono:

- un riferimento al Context dell'app. In questo caso, come spesso avviene, si risolve passando un riferimento **this** all'Activity stessa;
- il testo che si vuole mostrare. Ovviamente sarebbe il caso di sostituirlo con apposite

risorse stringa;

- la durata del messaggio utilizzando come valori uno di quelli messi a disposizione delle costanti della classe Toast: LENGTH_LONG e LENGTH_SHORT.

Da non dimenticare, l'invocazione del metodo `show()` senza la quale il Toast non apparirà.

Il risultato è visibile in figura:

Le Dialog

Immediatezza comunicativa e rapidità di implementazione sono i vantaggi principali del Toast ma ciò che offre spesso non basta. Arriva presto il momento di utilizzare delle vere finestre di dialogo.

Android offre la possibilità di avere Dialog grezze da configurare o in alternativa alcuni tipi già pronti che rispecchiano gli utilizzi più comuni: AlertDialog per gli avvisi, ProgressDialog per mostrare barre di progresso ed altre ancora.

Dialog è la superclasse di tutte le finestre di dialogo e ne rappresenta il tipo più duttile ma che lascia più lavoro

al programmatore. L'esempio seguente mostra codice che può essere eseguito all'interno di un metodo dell'Activity:

```
Dialog d=new Dialog(this);
d.setTitle("Login");
d.setCancelable(false);
d.setContentView(R.layout.dialog);
d.show();
```

Ciò che succede è descritto qui di seguito:

- istanziamo un oggetto di classe Dialog passando un riferimento al Context;
- impostiamo un titolo,

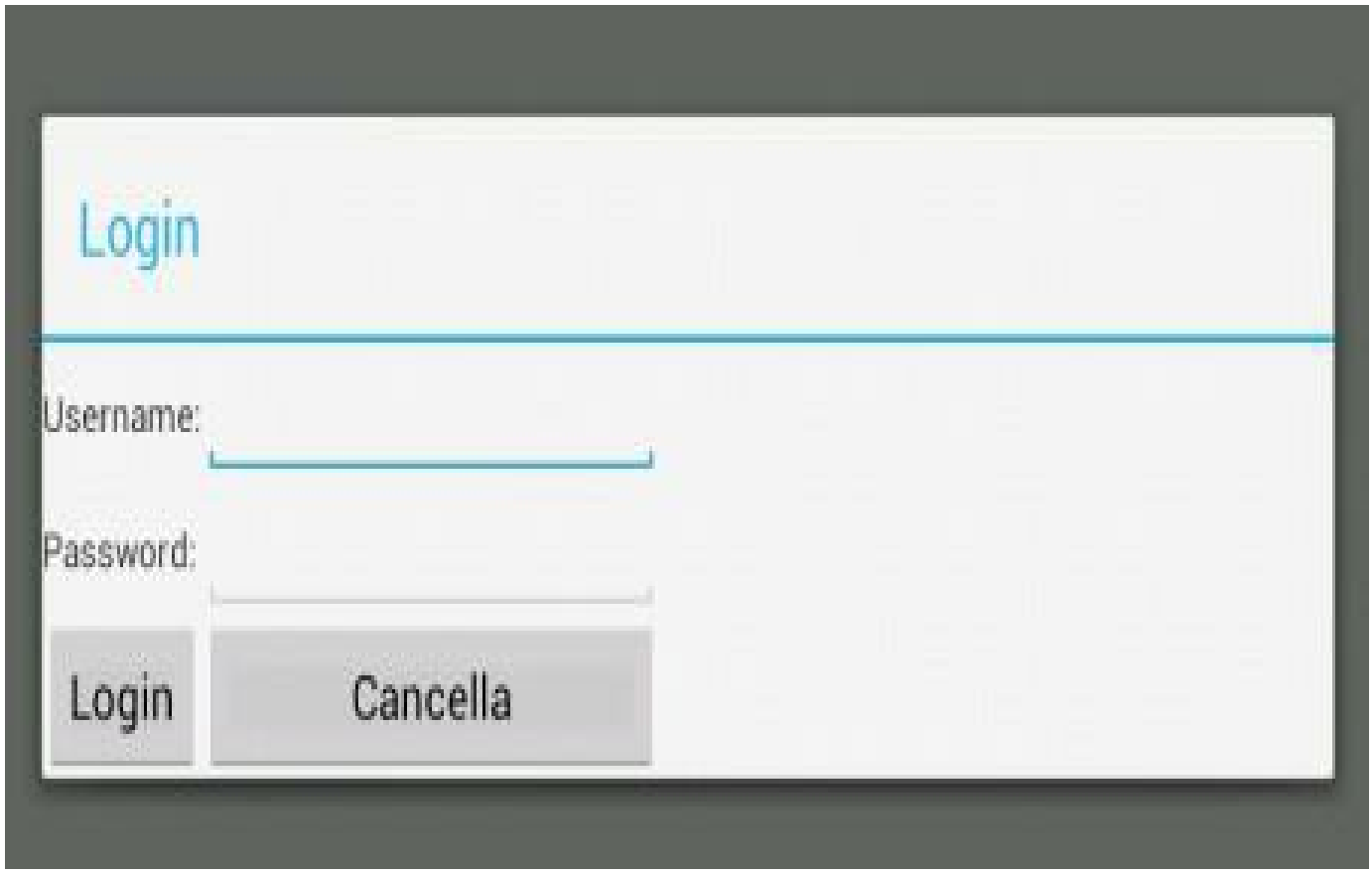
già proprietà della finestra stessa;

- definiamo la finestra come *modale* richiedendo la sua non-cancellabilità mediante `setCancelable`. Ciò impedirà che toccando il display al di fuori della finestra di dialogo essa si chiuda;
- assegnamo un layout alla finestra di dialogo come faremmo per un'Activity. Il layout si trova , come tutti i layout, in *res/layout* e contiene un normale form. Per

l’esempio si può usare lo stesso visto per i widget;

- ultimo ma non meno importante, invochiamo il metodo `show()` senza il quale la finestra di dialogo non apparirà.

Questo il risultato:



Il click sui pulsanti sarà gestito mediante click listener. Allo scopo è importante dotare, nel layout, i pulsanti di appositi id. Immaginando che il pulsante con l'etichetta "Login" abbia come id

`R.id.login`:

final Dialog d=new Dialog(this);

```
/*
 * OMISSIS: configurazioni varie della finestra di dialogo come le precedenti
 * */
Button b=(Button) d.findViewById(R.id.login);
b.setOnClickListener(new OnClickListener()
{
  @Override
  public void onClick(View arg0)
  {
      Toast.makeText(d.getContext(), "cliccato", Toast.LENGTH_LONG).show();
  }
});
d.show();
```

L’utilizzo del codice non stupisce ma

si notino comunque due particolarità:

- il Context necessario nel Toast è stato recuperato dal Dialog stesso;
- il Dialog dichiarato, a differenza dello snippet precedente, è stato etichettato come *final* in quanto, per una regola propria del linguaggio Java, dichiarato nello stesso metodo in cui viene istanziato l'oggetto anonimo che lo usa.

Dialog già “pronte”

Come si è visto, l’uso della classe Dialog non è proibitivo ma richiede comunque alcune operazioni. Parallelamente, Android offre tipi di finestre di dialogo, molto comuni, praticamente pronte all’uso.

I principali:

- **AlertDialog**: la più adattabile. Sfrutta una classe interna detta Builder per configurare i vari aspetti:

AlertDialog.Builder builder=new AlertDialog.Builder(this);

```
builder.setTitle("Attenzione!");
builder.setMessage("Operazione non valida!");
builder.show();
```

L'esempio di codice porta alla realizzazione di una finestra di dialogo che semplicemente mostra un titolo ed un messaggio: praticamente una normalissima AlertBox. AlertDialog inoltre ha già tre pulsanti innestati denominati PositiveButton, NegativeButton e NeutralButton per i quali è necessario solo definire le azioni di gestione del click. Infine piuttosto che richiamare lo show direttamente sul Builder, come mostrato, si può ottenere un riferimento alla finestra di dialogo

prodotta tramite il metodo createDialog.

- **ProgressDialog** è un derivato della AlertDialog pensato per lo più per ospitare indicatori di progresso sia in stile orizzontale (barra) che spinner (circolare). Può essere utilizzato in maniera molto agevole grazie alla versione statica del metodo show:

ProgressDialog progress = ProgressDialog.show(this, "Attendere", "Scaricamento in corso...", true);

- **DatePickerDialog** e **TimePickerDialog**, specializzati nella selezione di date e orari, mostrano controlli adatti e opportuni metodi per impostare le informazioni temporali oltre che rilevare gli eventi di selezione.

Capitolo 20 – Visualizzare pagine Web: WebView

Una WebView è un tipo di View che permette di visualizzare pagine web. La sua utilità principale è quella di permettere di **integrare una web application o più in generale un sito web nella propria applicazione**. Il motore della WebView risiede nella libreria WebKit già inclusa all'interno di Android per questo possiamo parlare di questo componente come di un browser vero e proprio in grado di eseguire Javascript e mostrare layout nella maniera più completa possibile.

Nella figura viene mostrato un semplicissimo esempio in cui il layout

dell'Activity è costituito esclusivamente dalla WebView e la si è usata per visualizzare direttamente il contenuto della pagina *http://www.html.it*.

Per ottenere questo risultato si sono compiute tre semplici operazioni:

1. si è creato un layout come il seguente:

```
<WebView
xmlns:android="http://schemas.android.com/apk/re
        android:id="@+id/webview"
        android:layout_width="match_parent
        android:layout_height="match_paren
/>
```

1. si è richiesto alla

WebView, già nell'onCreate dell'Activity, di caricare l'indirizzo remoto:

```
@Override
        protected void onCreate(Bundle savedInstanceState)
        {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main
             WebView myWebView = (WebView) findViewById(R.id.webview);
            myWebView.loadUrl("http://www.html.
        }
```

1. si è provveduto ad impostare nell'AndroidManifest.xml all'interno direttamente del nodo `<manifest>`, la permission per il collegamento ad Internet:

```
<manifest
xmlns:android="http://schemas.android.com/apk/re
        ...
        ...
                <uses-permission
android:name="android.permission.INTERNET"/>

        <application
            ...
```

Per la prima volta in questo corso, si incontra una **permission**. Rappresenta un aspetto di sicurezza che deve essere garantito ogni volta che l'applicazione vuole intraprendere attività di comunicazione o di interazione particolari. Le permission devono essere accettate dall'utente al momento dell'installazione e non nell'esecuzione a runtime. In questo caso, la nostra applicazione vuole accedere ad Internet per mezzo della WebView e per questo deve dichiararlo, semplicemente inserendo l'apposita permission nel manifest.

WebView e Javascript

Non si può parlare di Web al giorno d'oggi senza considerare Javascript. Di default, il suo uso in una WebView non è attivo. Per abilitare Javascript è sufficiente inserire queste righe di codice:

```
WebView myWebView = (WebView) findViewById(R.id.webview);

WebSettings webSettings = myWebView.getSettings();

webSettings.setJavaScriptEnabled(true);
```

Come si vede si è fatto uso delle *WebSettings* di WebView. Si tratta di un insieme di configurazioni che possono essere impostate per regolare tutti i comportamenti del componente quando

si trova a fronteggiare le classiche problematiche da browser:

- cache;
- font e layout;
- privacy e sicurezza.

La WebView può fare anche in modo che **Javascript interagisca con il codice Java** presente nell'applicazione. Per fare questo è necessario:

- creare il codice Java che vogliamo richiamare in Javascript apponendo l'annotazione

`@JavascriptInterface` sui metodi che devono essere visibili “da Javascript”:

```
public class WebAppInterface
{
        ...
        ...

        @JavascriptInterface
        public void executeJavaCode()
        {
            /*
                    Codice da attivare DA JAVASCRIPT
            */
```

```
        }
    }
```

registriamo presso la WebView il nostro oggetto Java, assegnandogli un'etichetta di nostra scelta, in questo caso AndroidObject:

```
WebView myWebView = (WebView) findViewById(R.id.webview);

myWebView.addJavascriptInterface(new WebAppInterface(), "AndroidObject");
```

Dopo questi passi, da Javascript potremo invocare il metodo `executeJavaCode` sull'oggetto Java che sarà stato registrato nella WebView con l'etichetta `AndroidObject`. Ad esempio, **nella pagina web che richiameremo dalla WebView potremo**

usare il seguente snippet Javascript:

```
<script type="text/javascript">
    function interactWithAndroid()
    {
        AndroidObject.executeJavaCode();
    }
</script>
```

Capitolo 21 – ListView e GridView

Tra le tante View di cui dispone Android, ne esiste una categoria particolarmente importante. Costituisce buona parte delle interfacce di cui sono dotate le app che usiamo. Stiamo parlando delle AdapterView e del loro rapporto con gli Adapter.

Adapter e AdapterView

Finora abbiamo apprezzato due modalità diverse per realizzare le varie parti delle nostre app:

- **design in XML**, per layout e risorse, dall'approccio piuttosto visuale ma orientato alla definizione di parti statiche;
- **sviluppo in Java**, fondamentale per le funzionalità dinamiche ma meno pratico per il disegno di porzioni di

layout.

Ma se dovessimo realizzare una “via di mezzo”: una visualizzazione iterativa di contenuti archiviati in strutture dati, potenzialmente variabili, come dovremmo comportarci? In fin dei conti, è un caso comunissimo, pensiamo alle app che mostrano liste di messaggi di vario genere, elenchi di notizie o risultati di una query su database.

Tutti questi lavori vengono affrontati con Adapter e AdapterView dove:

- **Adapter** è un componente collegato ad una struttura dati di

oggetti Java (array, Collections, risultati di query) e che incapsula il meccanismo di trasformazione di questi oggetti in altrettante View da mostrare su layout;

- **AdapterView** è un componente visuale che è collegato ad un adapter e raccoglie tutte le View prodotte dall'adapter per mostrarle secondo le sue politiche.

Passiamo subito ad un esempio pratico.

ListView, un AdapterView molto comune

Il primo esempio su questo argomento fondamentale viene fatto usando la più comune degli AdapterView, la ListView, ed il più immediato degli Adapter, l'ArrayAdapter.

l problema è: abbiamo un array di oggetti `String` e vorremmo che, iterativamente, una View ci mostrasse tutte le stringhe disposte in righe.

Agiamo così:

- otteniamo un riferimento alla struttura dati, in questo caso

l’array di String;

- istanziamo un ArrayAdapter assegnandogli, via costruttore, l’array che fungerà da sorgente dei dati e il layout da usare per ogni singola riga;
- recuperiamo la ListView predisposta nel layout e le assegnamo il riferimento all’adapter che sarà il suo “fornitore” di View.

Creiamo due layout:

- uno per l'Activity, nel file *activity_main.xml*, costituito solo dalla ListView:

```
<ListView
xmlns:android="http://schemas.android.com/apk/re
    android:id="@+id/listview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

- l'altro per la singola riga, in *row.xml*, contenente solo la TextView che mostrerà **il testo di ogni riga**:

```
<TextView
```

```
xmlns:android="http://schemas.android.com/apk/re
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
```

Il collegamento dinamico tra struttura dati/Adapter/Listview viene realizzato nell'`onCreate` dell'activity:

```
    @Override
            protected void onCreate(Bundle
savedInstanceState)
        {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.activity_main2);


                String[] citta=new String[]
{"Torino","Roma","Milano","Napoli","Firenze"};
            ArrayAdapter<String> adapter=new
ArrayAdapter<String>(this, R.layout.row,citta);
```

```
        ListView listView = (ListView) findViewById(R.id.listview);
        listView.setAdapter(adapter);
    }
```

L'immagine seguente mostra il risultato:

L'ArrayAdapter è il caso più

semplice di Adapter ma è comunque di uso molto frequente. Nella documentazione ufficiale è possibile trovare molti altri adapter già inclusi nel framework facilmente integrabili nelle proprie applicazioni.

GridView

Dire che la ListView nell'esempio precedente ha fatto molto poco non è del tutto sbagliato. La logica che trasforma gli oggetti in View è totalmente incluso nell'adapter. La ListView, o in generale gli AdapterView, si limitano in molti casi a recuperare View dall'adapter e a mostrarle nel layout.

Se volessimo sostituire la ListView con un altro AdapterView, diciamo la GridView specializzata in griglie, dovremmo compiere molto lavoro? Decisamente no.

Sarà sufficiente :

- **nel layout, sostituire la ListView con la GridView**, assegnando un numero di colonne alla griglia:

```
<GridView
xmlns:android="http://schemas.android.com/apk/r
    android:id="@+id/gridview"
    android:numColumns="3"
    android:layout_width="match_parent
    android:layout_height="match_paren
```

- nel codice Java semplicemente **sostituire l'uso della classe ListView con GridView**:

```
    @Override
                protected void onCreate(Bundle savedInstanceState)
        {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main2);
                        String[] nomi=new String[] {"Torino","Roma","Milano","Napoli","Firenze"};
                ArrayAdapter<String> adapter=new ArrayAdapter<String>(this, R.layout.row,nomi);

            // sostituiamo ListView con GridView
                    GridView gridView = (GridView) findViewById(R.id.gridview);
            gridView.setAdapter(adapter);
        }
```

La figura mostra come le View non saranno più mostrate in lista ma in una griglia a tre colonne.

Gestione degli eventi

Gli AdapterView hanno un altro ruolo molto importante: la gestione degli eventi. Come per tutte le View viene effettuato mediante il meccanismo dei listener. Un caso molto comune è la gestione del click su un elemento della lista, ciò che viene normalmente interpretato come selezione.

Tornando all'esempio della GridView, se volessimo far visualizzare un Toast che notifica quale elemento è stato selezionato dovremmo inserire il seguente codice:

```
GridView gridView = (GridView) findViewById(R.id.gridview);
gridView.setOnItemClickListener(new
```

```
OnItemClickListener()
    {
        @Override
         public void onItemClick(AdapterView<?>
av, View v, int pos,long id)
        {
                Toast.makeText(getApplicationContex
                        "Selezionato "+citta[pos],
Toast.LENGTH_LONG).show();
        }
    });
```

La classe OnItemClickListener viene utilizzata allo scopo di gestire il click ed il suo metodo onItemClick conterrà il vero codice da attivare ad ogni selezione di elemento. I suoi parametri in input conterranno tutte le informazioni utili tra cui:

- l’elemento su cui si è cliccato (il secondo, di classe View);
- la posizione che ricopre nella struttura (il terzo parametro, di tipo int). All’interno del metodo onItemClick si vede come l’informazione della posizione è stata sfruttata per recuperare dalla struttura dati l’oggetto (citta[pos]).

Proseguire lo studio di AdapterView e Adapter

Quello che è iniziato con questo capitolo è un argomento importante e molto articolato. Rappresenta un blocco fondamentale della UI Android.

Lo studio dell'argomento dovrà proseguire innanzitutto scoprendo i vari tipi di AdapterView disponibili nel framework. Nel prosieguo della guida ne verranno presentati altri tra cui, a brevissimo, lo Spinner ma per il resto la documentazione ufficiale rimarrà una fonte inestimabile di informazioni.

Altrettanto importante sarà apprezzare le potenzialità degli Adapter. In particolare, sarà fondamentale imparare

a realizzare un Adapter Custom in cui andremo a definire una logica personalizzata di trasformazione degli oggetti in View. Ciò sarà trattato in un capitolo lo successivo di questa guida.

Capitolo 22 – Spinner (menu a tendina)

Lo spinner è un altro widget molto comune, è il classico menu a tendina. In Andorid viene realizzato come AdapterView e tanto basta per farci comprendere il modo in cui dovremo usarlo.

Si è visto nelle lezioni precedenti che tutti gli AdapterView vengono grosso modo usati alla stessa maniera. È sufficiente collegare loro un Adapter che incapsula la logica di produzione delle View.

Per il resto l'AdapterView si occuperà di gestire gli eventi.

Spinner con valori fissi

Comunque lo Spinner trova la sua utilità anche in contesti meno complessi in cui si può usare come normale campo form per selezionare un valore in un dato insieme.

Pensiamo ad un form in cui si inseriscono i dati di una persona. Al momento di definire lo stato civile, la scelta ricade su un set di possibilità prestabilite: coniugato/a, divorziato/a, celibe/nubile, separato/a.

Il controllo ideale per effettuare questa scelta è senza dubbio lo Spinner . In questo caso, si potrebbe sentire meno il bisogno dell'Adapter in quanto la sorgente dati non cambierà più visto che

vengono annoverati già tutti gli stati civili possibili.

In questo caso, si può procedere agendo solo tra risorse XML:

- si crea un array di risorse stringa in un file della cartella *res/values* e lo si completa con tutti i valori necessari:

```
<string-array name="staticivili">
    <item>Divorziato/a</item>
    <item>Separato/a</item>
    <item>Coniugato/a</item>
    <item>Celibe/Nubile</item>
</string-array>
```

La risorsa sarà accessibile in XML mediante `@array/staticivili`

- nel file di layout in cui si trova lo Spinner si effettua una modifica. Si aggiunge l'attributo `android:entries` e gli si assegna la risorsa di stringhe a cui accedere:

```
<Spinner
    android:id="@+id/spinner"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:entries="@array/staticivili"/>
```

Seguendo questi due semplici passi,

nel layout troveremo lo spinner già popolato dei valori. Non è stato necessario apportare alcuna modifica nel codice Java e tantomeno istanziare direttamente un adapter.

Spinner con Adapter

Il comportamento dello Spinner legato ad un adapter è in buona parte uguale a quello della ListView e della GridView.

Vedremo ora un esempio che mostra un uso congiunto di Spinner e ListView in cui:

- lo Spinner mostra un elenco di Paesi;
- la ListView visualizza un elenco di città appartenenti tutte al Paese selezionato nello Spinner.

L’esempio ha anche il pregio di riepilogare molti concetti visti sinora nello studio delle GUI quindi lo si consideri un esercizio di validità generale.

L’immagine seguente mostra le varie fasi di funzionamento come appaiono in un emulatore.


1
Francia
Parigi
Lione
Marsiglia
2
Francia
Francia
Spagna
Italia
Spagna
Madrid
Barcellona
3

La fonte dei dati sarà una classe Java, molto semplice, che con liste e mappe fornirà i dati necessari all'esempio:

```
public class CountryList
{
                                        private
HashMap<String,ArrayList<String>> list;
        public CountryList()
        {
                            list=new  HashMap<String,
ArrayList<String>>();
                        ArrayList<String>  cities=new
ArrayList<String>();
                cities.add("Roma");
                cities.add("Torino");
                cities.add("Firenze");
```

```
        list.put("Italia", cities);
        cities=new ArrayList<String>();
        cities.add("Parigi");
        cities.add("Lione");
        cities.add("Marsiglia");
        list.put("Francia", cities);
        cities=new ArrayList<String>();
        cities.add("Madrid");
        cities.add("Barcellona");
        list.put("Spagna", cities);
    }
    public Collection<String> getCountries()
    {
        return list.keySet();
    }
                    public    Collection<String>
getCitiesByCountry(String c)
```

```
        {
            return list.get(c);
        }
    }
```

Il layout dell'Activity è molto semplice (file: *res/layout/activity_main.xml*), un RelativeLayout che mostra entrambe le View:

```
    <RelativeLayout
xmlns:android="http://schemas.android.com/apk/re
        xmlns:tools="http://schemas.android.com/tool
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <Spinner
          android:layout_width="@dimen/body_width
```

```
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="@dimen/margi
    android:id="@+id/countries"
    />

  <ListView
    android:layout_width="@dimen/body_width
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_below="@+id/countries"
    android:layout_marginTop="@dimen/margi
    android:id="@+id/cities"/>
</RelativeLayout>
```

mentre la forma che avrà la singola

riga dello Spinner e della ListView sarà la seguente (file: *res/layout/row.xml*):

```
<TextView
xmlns:android="http://schemas.android.com/apk/re
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="5dp"
        android:textSize="25sp"
        android:id="@+id/rowtext" />
```

Tenere presente che quando nel codice dell'Activity richiameremo l'id *R.id.rowtext* ci riferiremo alla TextView compresa in questo layout.

Il codice dell'Activity non offre grandi sorprese:

```
public class MainActivity extends Activity
{
        private CountryList countries=new CountryList();
        private ArrayAdapter<String> listviewAdapter;
        private ArrayAdapter<String> spinnerAdapter;

    @Override
        protected void onCreate(Bundle savedInstanceState)
    {
        // assegnazione del layout all'Activity
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main
```

```
                    // preparazione della ListView per
l'elenco delle città

                                ListView  lv=(ListView)
findViewById(R.id.cities);

                                    listviewAdapter=new
ArrayAdapter<String>(this, R.layout.row);

                lv.setAdapter(listviewAdapter);


                    // preparazione dello Spinner per
mostrare l'elenco dei Paesi

                                    spinnerAdapter=new
ArrayAdapter<String>(this, R.layout.row);

                spinnerAdapter.addAll(countries.getC

                                Spinner  sp=(Spinner)
findViewById(R.id.countries);

                sp.setAdapter(spinnerAdapter);

                    sp.setOnItemSelectedListener(new
OnItemSelectedListener()

                {
```

```
@Override
public void onItemSelected(AdapterView<?> arg0, View arg1,
int arg2, long arg3) {
TextView txt=(TextView) arg1.findViewById(R.id.rowtext);
String s=txt.getText().toString();
updateCities(s);
}

@Override
public void onNothingSelected(AdapterView<?> arg0)
{ }
});
}
```

```
    private void updateCities(String city)
    {
                                ArrayList<String>  l=
(ArrayList<String>)
                                    countries.getCitie
        listviewAdapter.clear();
        listviewAdapter.addAll(l);
    }

}
```

Da notare comunque che:

- essendo entrambi AdapterView, si sono

svolte le stesse operazioni per Spinner e ListView. In entrambi si è preparato un Adapter che gli è stato collegato con il metodo `setAdapter`;

- la gestione degli eventi è stata usata solo per lo Spinner. Avviene nella maniera classica illustrata nel capitolo delle View;
- all'interno del metodo `onItemSelected` che gestisce la selezione di una voce dello Spinner viene invocato il metodo

`updateCities` che aggiorna mediante l'adapter della ListView l'elenco delle città. Viene fatto in maniera molto semplice ricaricando una nuova lista.

Capitolo 23 – Stili e temi

Cos'è che trasforma questa lista:

in quest'altra ?

Risposta: l'applicazione di uno **stile**.

Tra le due immagini non ci sono differenze "strutturali". Si tratta della medesima combinazione di ListView e ArrayAdapter usata nei capitoli precedenti.

Il layout usato per raffigurare la singola riga è il seguente:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
xmlns:android="http:/schema.android.com/apk/res/
    android:layout_width="match_parent"
        android:layout_height="match_parent"/>
```

Quindi anche questo molto elementare.

In questo capitolo impareremo ad usare gli stili nelle interfacce Android. A livello concettuale, non si tratta altro che del porting dei CSS all'interno del framework e solitamente questo è ancora un elemento che fa contenti gli sviluppatori web.

Definire uno stile

Uno stile è una risorsa. Pertanto deve riuscire a trovare il suo posto all'interno della cartella res. Normalmente gli stili vengono ospitati nella sottocartella *res/values* e codificati mediante il tag XML `<style>`.

Lo stile che ha trasformato la lista delle figure precedenti è questo:

```
<resources
xmlns:android="http://schemas.android.com/apk/re
    <style name="rowstyle">
                                    <item
name="android:textColor">#FFFFFF</item>
                                    <item
name="android:textSize">25sp</item>
                                    <item
```

```
name="android:textStyle">bold</item>
                                    <item
name="android:background">@drawable/row_ba
                                    <item
name="android:paddingLeft">15dp</item>
    </style>
</resources>
```

È contenuto nel file *res/values/row_style* e come si vede dimostra che la creazione di uno stile è abbastanza intuitiva:

- si crea un nodo `<style>`;
- si assegna un attributo name per il nodo

`<style>` e questo diverrà il nome dello stile nel progetto;

- per ogni aspetto dello stile che si vuole curare si aggiunge un sottonodo `<item>` in cui l'attributo name definirà l'aspetto cui assegnare un valore ed il contenuto del nodo costituirà il valore assegnato.

Nell'esempio ciò che interessava era:

- rendere il testo di

colore bianco, in grassetto e leggermente più grande del normale:

```
<item
name="android:textColor">#FFFFFF</item>
<item name="android:textSize">25sp</item>
<item name="android:textStyle">bold</item>
```

- distanziarlo a sinistra dal bordo aggiungendo un po' di padding che potremmo definire un margine interno all'elemento:

```
<item
name="android:paddingLeft">15dp</item>
```

- dotare ogni TextView di angoli leggermente arrotondati e di un colore di sfondo sfumato dal celestino chiaro ad un azzurro non troppo scuro:

<item name="android:background">@drawable/row_ba

Per quanto riguarda i primi due punti non c'è molto da dire. Infatti in questi casi si deve solo cercare nella documentazione il nome dell'attributo che regola un aspetto e dimensionarlo appositamente, ad esempio `textColor`

rappresenta il colore del testo e gli assegnamo il valore esadecimale che rappresenta il bianco. Da notare che come **unità di misura del font** non è stato usato il `dp` ma `sp`. Il concetto alla base di `sp` è identico a quello dei `dp` ma è più rispettoso delle problematiche dei font.

Ciò che è particolare è il terzo punto, l a **creazione dello sfondo**. Come si vede rimanda ad un'altra risorsa, di tipo drawable, e di nome *row_background*. Per fare ciò creeremo un file in una cartella drawable, ad esempio: *res/drawable-mdpi/row_background.xml*.

Ecco il suo contenuto:

```
    <shape
xmlns:android="http://schemas.android.com/apk/re
>
        <gradient
            android:startColor="#2669DE"
            android:endColor="#99ADD1"
            android:angle="90"/>
        <corners android:radius="5dp"/>
    </shape>
```

Contiene un nodo <shape> che, anche se può sembrare strano, serve a **disegnare in XML**. Uno shape è una forma, di default rettangolare, che può essere configurata mediante i suoi sottonodi.

Questo shape ha due sottonodi:

- `<gradient>` che realizza la sfumatura definendo i codici esadecimali dei colori di partenza e di arrivo;
- `<corners>` indica di quanto devono essere arrotondati gli angoli.

Questo del disegno in XML è uno dei settori più vasti in assoluto tra le risorse Android quindi è necessario studiare la documentazione per ulteriori approfondimenti. Ciò che conta qui è dimostrare come la creazione di uno stile possa essere rapida.

Appena creati questi due file (*res/values/row_style.xml* e *res/drawable-mdpi/row_background.xml*) si può **applicare il nuovo stile**, di nome rowstyle, alla TextView del layout precedente semplicemente assegnando il nome dello stile all'attributo style:

```
<TextView
xmlns:android="http://schemas.android.com/apk/re
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    style="@style/rowstyle"/>
```

Ereditarietà tra stili

Un aspetto importante degli stili Android è **l'ereditarietà**. Non dobbiamo necessariamente partire da zero nel creare uno stile ma possiamo renderli, per così dire, uno il derivato di un altro.

L'esempio precedente potrebbe essere rivisto così:

```
<resources
xmlns:android="http://schemas.android.com/apk/re

<style name="rowstyle"
parent="@style/rowbackground">

<item
name="android:textColor">#FFFFFF</item>

<item
name="android:textSize">25sp</item>
```

```
                <item
name="android:textStyle">bold</item>
        </style>

        <style name="rowbackground">
                <item
name="android:background">@drawable/row_ba
                <item
name="android:paddingLeft">15dp</item>
        </style>
    </resources>
```

Esistono ora due stili:

- `rowbackground` che gestisce solo il background;
- `rowstyle` che contiene le proprietà relative al testo ma con l'attributo parent dichiara di essere "figlio di rowbackground". Per questo rowstyle includerà in sé tutte le valorizzazioni espresse nello stile-padre.

Il vantaggio di ciò è che si può

stratificare la creazione di stili evitando di ripetere configurazioni simili in stili diversi. Potremmo, ad esempio, creare un altro stile per il testo ma sempre figlio di rowbackground. Questo nuovo stile creerebbe testo diverso da rowstyle ma con lo stesso sfondo.

Temi

Per tema, si intende in Android nient'altro che **uno stile applicato ad un'Activity o all'intera applicazione**. Il modo in cui si fa questo consiste nell'inserire l'attributo `android:Theme` nel file *AndroidManifest.xml* all'interno di un nodo `<activity>` o `<application>`.

Il valore di `android:Theme` sarà l'id di una risorsa stile definita come precedentemente spiegato.

Capitolo 24 – Creare Custom Adapter

L'Adapter è un pattern già presentato in questo libro. È il meccanismo per impostare agevolmente la visualizzazione di oggetti Java su un layout di un'app Android.

Finora le accoppiate AdapterView-Adapter utilizzate negli esempi sono state ListView-ArrayAdapter o GridView-ArrayAdapter.

Di Adapter ne esistono molti nel framework. La documentazione offre tutti i dettagli in merito ma spesso capita di aver bisogno di **creare una visualizzazione personalizzata**.

In questi casi, si può realizzare un Adapter in versione *custom* ed in questo capitolo vedremo come. Studiare questa casistica fornisce il programmatore non solo di uno strumento utilissimo, ma anche di un'esperienza formativa molto significativa che permette di osservare il funzionamento di un Adapter "dall'interno".

Nell'esempio che andremo ad utilizzare, un'Activity mostra un elenco di articoli. Potrebbe essere l'interfaccia di un NewsReader ma qui verrà trattata in maniera simulata. Gli oggetti Java sono prodotti da un metodo di supporto interno all'Activity.

Ogni articolo, nel progetto, è rappresentato da un oggetto di classe

ArticlesInfo i cui membri rappresentano, rispettivamente, il titolo dell'articolo, la categoria tematica di appartenenza e la data di pubblicazione:

```
public class ArticleInfo
{
        private String title;
        private String category;
        private Date date;
        /*
         * OMISSIS: la classe possiede tutti i setter e i getter
        * per gestire i membri privati
        *
        * */
}
```

Affinchè ognuno di questi oggetti, per così dire, si trasformi in una riga della ListView contenuta nell'Activity **prepariamo subito un layout** nel file *res/layout/listactivity_row_article.xml*::

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/re
    android:layout_width="match_parent"
    android:layout_height="200dp"
    android:background="@drawable/row_back
    android:descendantFocusability="blocksDes
    android:padding="10dp">
    <LinearLayout
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true"
```

```
    android:orientation="vertical"
    android:id="@+id/ll_text"
      android:layout_toLeftOf="@+id/btn_boo
      <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="@style/big_textstyle"
    android:id="@+id/txt_article_description"
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="@style/small_textstyle"
    android:id="@+id/txt_article_url"/>
</LinearLayout>
<TextView
    android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:maxLength="5"
        style="@style/small_textstyle"
        android:layout_alignParentRight="true"
        android:layout_alignParentTop="true"
        android:id="@+id/txt_article_datetime"/>
</RelativeLayout>
```

Nel layout appena riportato sono presenti riferimenti a stili e drawable di sfondo di cui non viene offerto qui il listato ma che ricalcano perfettamente ciò che è stato spiegato in merito in questa guida. **Importante ricordare che questo layout, rappresenta la forma di una singola riga che apparirà nell'AdapterView**.

L'elemento che si occuperà della trasformazione di ogni oggetto ArticlesInfo in una View sarà proprio l'Adapter: ma non uno standard, uno personalizzato creato da noi.

Per fare questo dobbiamo:

1. Creare una classe che chiameremo `ArticlesAdapter`, estensione di `BaseAdapter`;
2. Fare in modo che la classe possieda un riferimento alla struttura dati da visualizzare, magari passato tramite

costruttore. Nel nostro caso sarà una List `<ArticlesInfo>`;

3. **Implementare obbligatoriamente i metodi astratti** di BaseAdapter:

-

Vediamo il codice dell'Adapter:

```
{

        private List<ArticleInfo> articles=null;

        private Context context=null;

            private SimpleDateFormat simple=new
SimpleDateFormat("dd/MM",Locale.ITALIAN);


                    public ArticlesAdapter(Context
context,List<ArticleInfo> articles)

        {

            this.articles=articles;

            this.context=context;
```

```
}

@Override
public int getCount()
{
        return articles.size();
}

@Override
public Object getItem(int position)
{
        return articles.get(position);
}

@Override
public long getItemId(int position)
```

```
        {
            return getItem(position).hashCode();
        }

        @Override
          public View getView(int position, View v,
ViewGroup vg)
        {
            if (v==null)
            {
                v=LayoutInflater.from(context).in
null);
            }
                        ArticleInfo ai=(ArticleInfo)
getItem(position);
                        TextView txt=(TextView)
v.findViewById(R.id.txt_article_description);
            txt.setText(ai.getTitle());
```

```
                txt=(TextView)
v.findViewById(R.id.txt_article_url);
                txt.setText(ai.getCategory());
                txt=(TextView)
v.findViewById(R.id.txt_article_datetime);
                txt.setText(simple.format(ai.getDate()))
                return v;
        }

    }
```

Come si può vedere, i metodi non sono particolarmente complicati ma su **getView** vale la pena soffermarsi un attimo.

Al suo interno, per prima cosa, viene controllato se la View passata in input è

nulla e solo in questo caso viene inizializzata con il `LayoutInflater`. Questo aspetto è molto importante ai fini della salvaguardia delle risorse infatti Android riciclerà quanto possibile le View già create. Il LayoutInflater attua per i layout quello che abbiamo già visto fare per i menu con il `MenuInflater`. In pratica la View da creare verrà strutturata in base al "progetto" definito nel layout XML indicatogli.

Dopo il blocco `if`, la View non sarà sicuramente nulla perciò procederemo al completamento dei suoi campi. I dati verranno prelevati dall'oggetto ArticleInfo di posizione position recuperato mediante `getItem`, già implementato. Al termine, `getView`

restituirà la View realizzata.

Questo Adapter incarnerà tutta la logica di trasformazione infatti per il resto l'Activity è molto semplice. Tra l'altro, estende la classe ListActivity che ha un layout costituito da una ListView e alcuni metodi per la sua gestione:

- `getListView()` per recuperare un riferimento alla ListView;
- `setListAdapter()` e `getListAdapter()` per ottenere accesso all'Adapter.

```
public class MainActivity extends ListActivity
{
        private ArticlesAdapter adapter=new
ArticlesAdapter(this, generateNews());

    @Override
        protected void onCreate(Bundle
savedInstanceState)
    {
        super.onCreate(savedInstanceState);
          getListView().setPadding(10, 10, 10,
10);
        setListAdapter(adapter);
    }

    private List<ArticleInfo> generateNews()
    {
```

```
                ArrayList<ArticleInfo> list=new ArrayList<ArticleInfo>();
                Calendar c=Calendar.getInstance();

                ArticleInfo tmp=new ArticleInfo();
                tmp.setTitle("WordPress: integrare un pannello opzioni nel tema");
                tmp.setCategory("CMS");
                c.set(2014,3,23);
                tmp.setDate(new Date(c.getTimeInMillis()));
                list.add(tmp);
              /*
                    * OMISSIS: il codice crea altri oggetti "fittizi" da visualizzare
                 * */
                return list;
        }
```

```
}
```

Come si è visto la realizzazione di un Adapter personalizzato non presenta eccessive difficoltà ma offre grandi potenzialità. Nel layout che struttura il singolo item dell'AdapterView si può inserire qualunque controllo visuale e ciò evita che il programmatore si trovi costretto a scendere a compromessi per la realizzazione della propria interfaccia utente.

Infine, nell'esempio si è fatto uso di una ListView ma un Adapter custom può lavorare con qualunque AdapterView.

Capitolo 25 – Fragment in Android

I Fragment costituiscono senz'altro uno dei più importanti elementi per la creazione di una interfaccia utente Android moderna. Il loro ruolo a partire da Android 3.0 è diventato preponderante tanto che ormai rappresentano una delle conoscenze più importanti per il programmatore.

Un Fragment è una porzione di Activity. Ma si faccia attenzione a comprenderne bene il ruolo. Non si tratta solo di un gruppo di controlli o di una sezione del layout. Può essere definito più come una specie di sub-activity con un suo ruolo funzionale

molto importante ed un suo ciclo di vita.

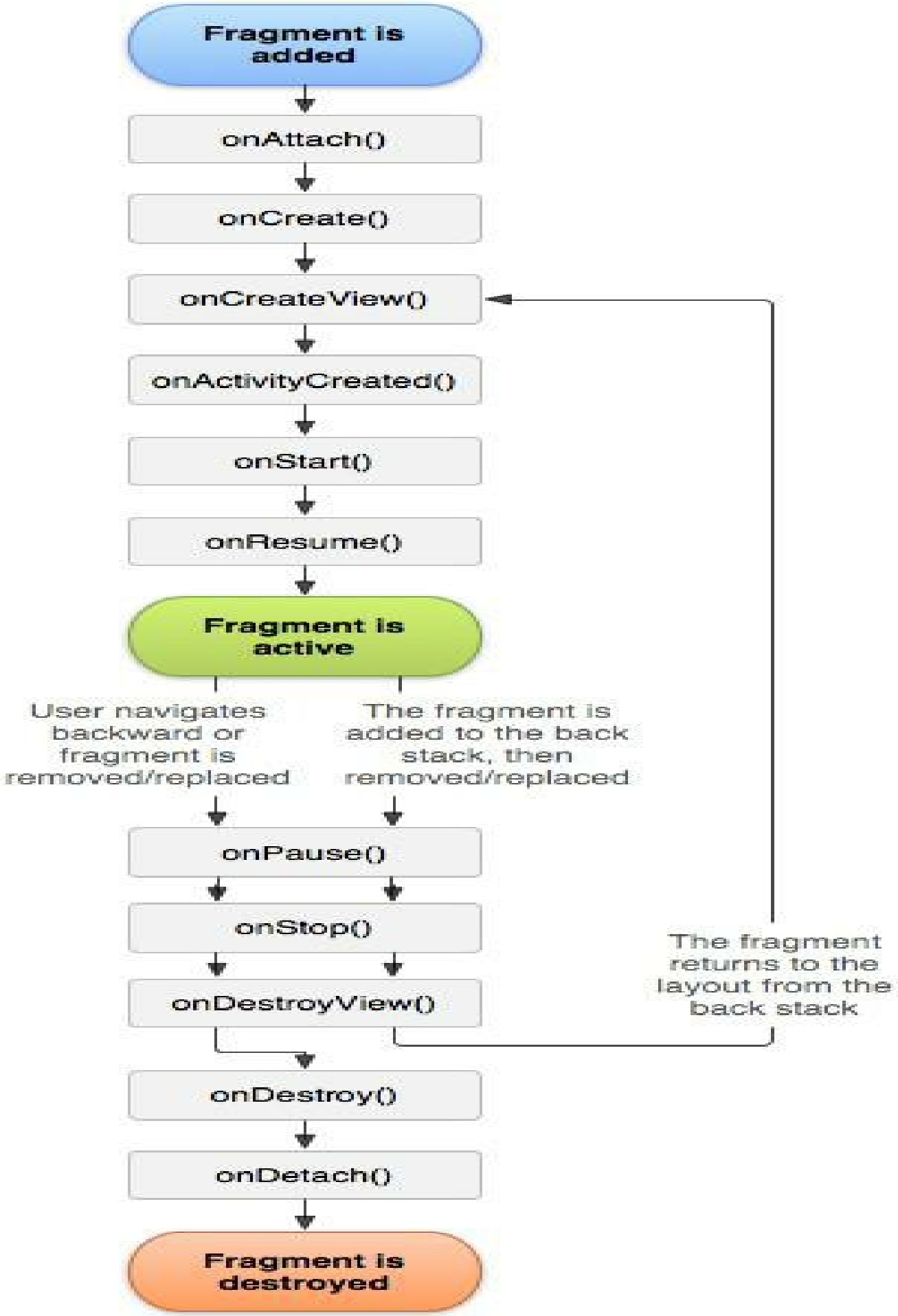
Fragments e Activity

Definiamo subito il rapporto tra Fragments e Activity.

Prima di tutto un Fragment non può vivere senza un'Actvity. Tipicamente nei nostri programmi creeremo più Fragments che si alterneranno nel layout mentre di Activity ne sarà sufficiente una (ma possono essere anche di più).

Come detto il Fragment ha il suo ciclo di vita fortemente collegato con quello dell'Activity di appartenenza.

La figura qui riportata mostra la sequenza di stati che scandiscono la vita del Fragment.


Fragment is added
onAttach()
onCreate()
onCreateView()
onActivityCreated()
onStart()
onResume()
Fragment is active
User navigates backward or fragment is removed/replaced
The fragment is added to the back stack, then removed/replaced
onPause()
onStop()
onDestroyView()
The fragment returns to the layout from the back stack
onDestroy()
onDetach()
Fragment is destroyed

Come si vede ricordano molto quelli dell'Activity.

La fase più variegata è l'inizializzazione del fragment:

- **onAttach**: segnala il momento in cui il Fragment scopre l'Activity di appartenenza. Attenzione che a quel punto l'Activity non è stata ancora creata quindi si può solo conservare un

riferimento ad essa ma non interagirvi;

- **onCreate**: è la creazione del Fragment in quanto componente;
- **onCreateView**: il programmatore vi lavorerà spesso. È il momento in cui viene creato il layout del Fragment. Solitamente qui si fa uso del *LayoutInflater*;
- **onActivityCreated**: segnala che la creazione dell'Activity è stata completata, vi si può

interagire in tutto e per tutto.

Gli altri metodi di callback del ciclo di vita vengono chiamati in corrispondenza degli omonimi metodi dell'Activity.

Hello Fragment!

Come abbiamo fatto per le Activity, anche per i Fragment inizieremo con un "Hello World". In questo caso, lo scopo dell'esempio non sarà, ovviamente, tanto l'apparizione del messaggio di saluto quanto osservare le fasi che portano alla creazione di un Fragment e del suo innesto all'interno di un layout.

Per raggiungere lo scopo, seguiremo questi *step* che tratteranno sia XML, i primi due, che Java, i secondi due :

- creeremo il layout per l'Activity in cui ricaveremo il posto per

alloggiare il nostro Fragment;

- definiremo il layout del Fragment che conterrà il vero aspetto dell'Activity e, di conseguenza, anche la stringa "Hello World";
- definiremo la classe Fragment che essenzialmente servirà a caricare il layout di cui al punto precedente;
- creeremo la nostra Activity che svolgerà per lo più il ruolo di bacino di Fragment.

Il primo frammento di codice mostra il layout del Fragment (file: *res/layout/fragment_main.xml*).

Come si vede, se fosse stato destinato ad un'Activity sarebbe stato identico. Quindi la novità architetturale dei Fragment non influenza il layout.

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/a
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/activity_vertical_


    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
```

```
</RelativeLayout>
```

l layout dell'Activity è il seguente (file: *res/layout/activity_main.xml*):

```
<FrameLayout
    xmlns:android="http://schemas.android.com/a
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Tutto lo spazio disponibile verrà riempito da un layout che non abbiamo mai usato sinora: il **FrameLayout**.

Viene utilizzato quando vi si deve ospitare un unico elemento, in questo caso il Fragment. Fondamentale definire l'id in quanto questo Layout svolgerà il

ruolo di contenitore del Fragment e pertanto verrà invocato dal codice Java.

La classe Fragment mostra evidenti alcune caratteristiche:

- estende la classe Fragment del framework;
- presenta metodi propri del ciclo di vita dei Fragment. Come già accennato, sarà frequente l'ovverride del metodo `onCreateView` in quanto è il momento in cui viene allestita l'interfaccia utente mostrata dal Fragment. Non ci

sorprendono (ormai) le operazioni svolte al suo interno: assegnazione di un Layout mediante LayoutInflater.

```
public class HelloFragment extends Fragment
{
    public HelloFragment() {
    }

    @Override
     public View onCreateView(LayoutInflater inflater, ViewGroup container,
                Bundle savedInstanceState)
     {
                View rootView = inflater.inflate(R.layout.fragment_main, container,
```

```
false);
            return rootView;
        }
    }
```

Il codice dell'Activity contiene solo il metodo `onCreate`. Al suo interno, vengono svolte nelle prime due righe le operazioni consuete ma questa volta il caricamento del layout con `setContentView` non basta. Infatti questo porterà a display solo il FrameLayout ancora vuoto

Per aggiungere il Fragment, si procederà per via dinamica richiedendo

al FragmentManager l'avvio di una transazione `add` che aggiungerà il nuovo Fragment di classe HelloFragment al layout identificato da `R.id.container`.

```
public class MainActivity extends ActionBarActivity
    {
        @Override
            protected void onCreate(Bundle savedInstanceState)
        {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main
            if (savedInstanceState == null)
            {
                getSupportFragmentManager().b
```

```
                .add(R.id.container,
new HelloFragment()).commit();

            }
        }
    }
```

L'operazione `add` fa parte delle **FragmentTransactions.** Le useremo anche nel prossimo capitolo ma intanto si pensi ad esse come delle operazioni che devono essere svolte dal FragmentManager sui Fragments amministrati.

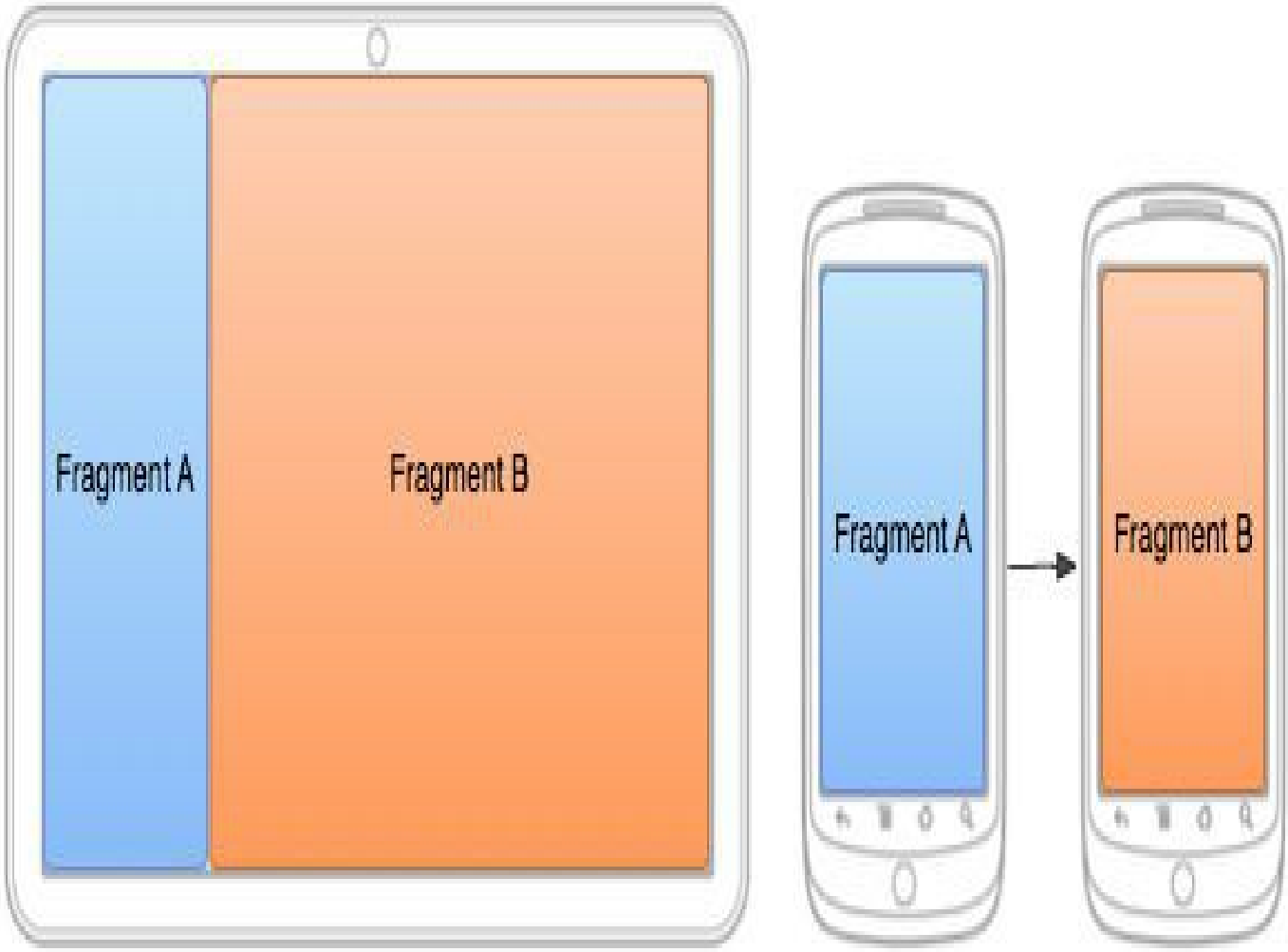
Le FragmentTransactions più comuni sono:

add	Aggiunge un Fragment all'Activity
remove	Rimuove un fragment precedentemente aggiunto
replace	Sostituisce un Fragment con un altro
hide	Nasconde un fragment
show	Mostra un fragment precedentemente nascosto

Notare ancora che, come avviene per le transazioni nei database, le operazioni iniziano con un `beginTransaction` e vengono definitivamente salvate con un `commit`.

Capitolo 26 – Layot adattabili Smartphone/Tablet in Android

I Fragments possono essere visti come blocchi componibili che permettono di rendere il layout adattabile al dispositivo. Se la frammentazione dei dispositivi rappresenta una problematica di rilievo per i programmatori Android, i Fragments rappresentano in buona parte una soluzione.



La figura mostra due dispositivi di tipo diverso ed in configurazioni differenti:

- uno **smartphone in portrait**. Immaginiamolo

con uno schermo piccolo, anche 3 pollici;

- un **tablet**, quindi schermo almeno da 7 pollici, posizionato **in landscape**.

I layout presenti su entrambi sono costituiti da due fragments, gli stessi due Fragment: FragmentA e FragmentB.

Con adeguate configurazioni delle risorse e qualche aggiunta al codice visto nel capitolo precedente possiamo creare anche noi **un layout adattabile** che riesca a mostrarsi in *one-pane* su smartphone e *two-pane* su tablet in landscape.

Configurazione delle risorse

Parlando delle risorse, avevamo accennato alla loro gestione multipiattaforma. È arrivato il momento di vederla al lavoro, costituirà il punto di partenza del nostro layout adattabile.

Creiamo **due cartelle di risorse layout**:

- *layout-large-land* che verrà usato solo per dispositivi con display large in posizione landscape;
- *layout,* la cartella di

default. Verrà chiamata in causa per tutte le altre situazioni.

La configurazione multipla ha successo se in entrambe le cartelle mettiamo il file di layout con il medesimo nome, *activity_main.xml*.

Da questo momento, l'Activity cercherà sempre la risorsa `R.layout.activity_main` ma questa, **in base alla configurazione del dispositivo**, corrisponderà ora al file *res/layout-large-land/activity_main.xml* ora al file *res/layout/activity_main.xml*.

Vediamo entrambi i file di layout.

File 1: *res/layout/activity_main.xml*:

```
<FrameLayout
    xmlns:android="http://schemas.android.com/a
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Questo primo layout è identico a quello visto nel capitolo precedente. È un FrameLayout che ospiterà un fragment singolo assegnato dinamicamente con FragmentTransactions.

File 2: *res/layout-large-land/activity_main.xml*:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/a
```

```
        android:layout_width="match_parent"
        android:layout_height="match_parent">
                                        <fragment
android:name="it.html.guida.gui.fragments.Countr
        android:id="@+id/countryfrag"
        android:layout_width="0dp"
          android:layout_height="match_parent"
        android:layout_weight="1"/>
                                        <fragment
android:name="it.html.guida.gui.fragments.CityFra
        android:id="@+id/cityfrag"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="2"/>
    </LinearLayout>
```

Nel layout two-pane, i Fragments

appaiono insieme sin dall'inizio mentre le FragmentTransactions non dovranno più intervenire. Come segnaposti, abbiamo i tag `<fragment>`. Il loro attributo android:name indica quale tipo di Fragment dovrà posizionarsi in ogni collocazione.

Comunicazione tra Fragments

L'Activity svolge il ruolo di snodo funzionale e di comunicazione tra i due Fragments sia che essi appaiano contemporaneamente sia che si alternino sul display.

Affinchè i Fragments siano riutilizzabili in più contesti è necessario che non si conoscano tra loro né che conoscano l'Activity alla quale, comunque, devono essere collegati. "Massima coesione, minimo accoppiamento": questo potrebbe essere lo slogan dei Fragments.

All'interno dei Fragment **non verrà**

mai menzionata esplicitamente la classe di appartenenza dell'Activity .

P i u t t o s t o **verrà definita un'interfaccia che sarà implementata dall'Activity**. Questa interfaccia costituirà il "protocollo" di comunicazione Fragment-Activity.

L'esempio: Paesi e città

L’esempio mette in pratica il classico modello master/detail. Il master è un fragment, classe CountryFragment, che mostra una lista di Paesi. La scheda detail invece è un CityFragment che mostra un elenco di città appartenenti al Paese selezionato nel master.

In base alle premesse iniziali, vogliamo che i Fragments si presentino accoppiati su schermi large in landscape e si alternino in tutti gli altri casi.

A scopo di esempio, la sorgente dati è fittizia. È totalmente contenuta in una classe CountryList, già usata nel capitolo sugli Spinner. Si invocheranno i metodi:

- `Collection<String> getCountries()`: restituisce l'elenco dei Paesi;
- `Collection<String> getCitiesByCountry(St country)`: restituisce un elenco di città situate nel Paese.

Selezionando un elemento nella lista dei Paesi, è necessario che l'elenco di città presente nell'altro fragment venga aggiornato. In tutto questo, l'Activity svolgerà il ruolo di mediatore.

```
public class CountryFragment extends ListFragment
{
    interface OnFragmentEventListener
    {
        void selectCountry(String c);
    }
        private OnFragmentEventListener listener=null;
    private CountryList l=new CountryList();
    private String[] countries=null;

    public CountryFragment()
    {
                                countries=new String[l.getCountries().size()];
        l.getCountries().toArray(countries);
```

```
        }
        @Override
        public void onAttach(Activity activity)
        {
                super.onAttach(activity);
                  listener=(OnFragmentEventListener)
activity;
        }
        @Override
          public View onCreateView(LayoutInflater
inflater, ViewGroup container,Bundle
savedInstanceState)
        {
                ArrayAdapter<String> adapter=new
ArrayAdapter<String>((Context)
listener,android.R.layout.simple_list_item_1,countri
                setListAdapter(adapter);
                  return super.onCreateView(inflater,
```

```
container, savedInstanceState);
        }
        @Override
            public void onListItemClick(ListView lv,
View v, int position, long id)
        {
            listener.selectCountry(countries[positi
        }
    }
```

Il CountryFragment mostra alcune particolarità:

- contiene un'interfaccia OnFragmentEventListener Se si osserva il metodo `onAttach` si vede che

tale interfaccia verrà usata per riferirsi all'Activity senza dover usare esplicitamente la sua classe;

- è stato esteso ListFragment che funziona in maniera simile al ListActivity. All'interno dell'`onCreateView` viene impostato l'Adapter;
- `onListItemClick` rappresenta il momento in cui si notifica all'Activity che è stato selezionato un Paese.

Il codice dell'Activity è il seguente:

```
public class MainActivity extends ActionBarActivity implements OnFragmentEventListener{


        @Override

                protected void onCreate(Bundle savedInstanceState)

        {

                super.onCreate(savedInstanceState);

                setContentView(R.layout.activity_main


                    if (findViewById(R.id.container) != null)

                {

                        // Se è presente il FrameLayout con id container,
```

```
                    // vuol dire che siamo in
SINGLE-PANE perciò
                    // è necessario aggiungere il
Fragment con la transazione.


                    // Se savedInstanceState non è
nullo, non siamo alla
                    // prima visualizzazione perciò
non serve aggiungere il Fragment.


                    if (savedInstanceState != null)
                        return;
                    getSupportFragmentManager().b
                            .add(R.id.containe
new CountryFragment()).commit();
                }
        }
        @Override
```

```
public void selectCountry(String c)
{
                    CityFragment cityFrag =
(CityFragment)
          getSupportFragmentManager().findFr

                if (cityFrag != null &&
cityFrag.isInLayout())
{
          // Il Fragment delle città è già nel
layout quindi
          // ne chiediamo solo l'aggiornamento.

     cityFrag.onSelectedCountry(c);
}
else
{
```

```
                // Siamo in SINGLE-PANE, quindi le
FragmentTransaction
                // operano lo switch tra Fragment.


                        CityFragment frag= new
CityFragment();
                Bundle b=new Bundle();
                b.putString("country", c);
                frag.setArguments(b);
                        FragmentTransaction
ft=getSupportFragmentManager().beginTransactio
                ft.replace(R.id.container, frag);
                ft.addToBackStack(null);
                ft.commit();
            }
            }
    }
```

L'Activity si troverà esplicitamente a dover gestire l'esatta composizione del layout. Se si notano i commenti nel codice si vede come si procede:

- `onCreate`: se nel layout è reperibile il layout con id `R.id.container` si è per forza in one-pane;
- `selectCountry`: se si trova presente nel layout un CityFragment allora siamo in two-pane altrimenti si fa uso delle FragmentTransaction per

reperirlo.

L'ultimo codice da mostrare è il CityFragment:

```
public class CityFragment extends ListFragment
{
                private ArrayAdapter<String> adapter=null;
        private CountryList l=new CountryList();

        public CityFragment() {
        }

        @Override
                public void onActivityCreated(Bundle savedInstanceState)
        {
```

```
        super.onActivityCreated(savedInstanc
         adapter=new ArrayAdapter<String>
(getActivity(),android.R.layout.simple_list_item_1);
        setListAdapter(adapter);
        Bundle b=getArguments();
        if (b!=null)
        {
            String c=b.getString("country");
            onSelectedCountry(c);
        }
    }

    public void onSelectedCountry(String
country)
    {
        adapter.clear();
        adapter.addAll(l.getCitiesByCountry(c
```

```
        }
    }
```

Al suo interno, la richiesta di aggiornamento del layout viene fatta mediante l'invocazione di `onSelectedCountry`. Notare inoltre l'utilizzo degli *arguments* per passare valori nella comunicazione Fragment-Activity.

Conclusioni

Sicuramente questo è un esempio molto basilare ma che vuole fornire i rudimenti per poter creare applicazioni più complesse rese flessibili dall'attuazione delle regole viste sinora:

- Fragments collegati tra loro il minimo possibile per permetterne un maggior riutilizzo;
- Activity usata per lo più come snodo di comunicazione tra Fragments;
- risorse in grado di

saparare all'origine i layout distinguendo con id diversi i contenitori dei Fragments;

- ogni classe Fragment deve avere sin dalla nascita uno scopo ben preciso e contenere in sé tutta la logica necessaria per raggiungerlo in maniera indipendente.

Capitolo 27 – Gestire i file

Tutte le applicazioni Android hanno la possibilità di interagire con il filesystem delle memorie di massa installate nel dispositivo, spesso costituite da SD card.

In questo capitolo vedremo come possono essere gestite sia nel caso di supporti fissi che rimovibili. Il requisito fondamentale per il loro utilizzo è una **buona conoscenza del sottosistema di I/O del linguaggio Java**, specialmente del suo concetto fondante, gli Stream.

Se il lettore ha già confidenza con questi strumenti non troverà grosse difficoltà nel seguire il discorso ed i principali aspetti di novità che potrà

riscontrare saranno gli accorgimenti presi per l'adattamento di queste tecniche alla filosofia del sistema Android.

Distinzione basilare: Internal Storage vs. External Storage

La prima distinzione da imparare è quella tra lo spazio interno all'applicazione e quello esterno. Parte di questa distinzione verte su quel concetto di riservatezza dei dati propri di un'applicazione di cui si è discusso al principio di questa guida.

Ogni app ha a disposizione uno spazio disco, detto Storage interno (o Internal Storage) che risiede in una parte del filesystem e a cui solo l'applicazione dovrebbe accedere. Il percorso in Android che porta in questa posizione è */data/data/package_java_della_applica*

Il percorso è in formato Linux quindi lo '/' iniziale indica la root del filesystem. Notare che in */data/data* lo spazio dell'applicazione prende il nome dal package Java. Se ad esempio la nostra app avesse package Java *it.html.guida*, i file salvati nello storage interno sarebbero collocati in: */data/data/it.html.guida/files*.

Per External Storage, Android sceglie una posizione in cui collocare tutto un insieme di risorse che sono di utilità e consultazione generale nel dispositivo (musica, suonerie, film e così via). Solitamente dovrebbe essere collocato su una SD card esterna al sistema ma non è detto che sia un supporto rimovibile, potrebbe essere interno al

dispositivo e fisso.

Ora si entrerà nei dettagli di entrambe le tipologie di storage.

Storage interno

Per accedere allo storage interno si usano per lo più due metodi, entrambi appartenenti al Context:

- `FileInputStream` **openFileInput**`(String filename)`: apre un file in lettura. Come parametro in input viene passato il nome del file. Non serve specificare il percorso in quanto sarà obbligatoriamente quello messo a disposizione dallo storage interno;

- `FileOutputStream` **openFileOutput**`(String filename, int mode)`: apre uno stream in scrittura anche questo nello storage interno. Per il nome del file, vale quanto detto per l'input. A proposito del secondo parametro, si può impostare alla costante Context.MODE_APPEND per concatenare i nuovi contenuti a quelli già esistenti nel file.

Notare che sono disponibili anche due costanti *MODE_WORLD_READABLE* e

MODE_WORLD_WRITEABLE che servono a rendere il file accessibile anche al di fuori dello storage interno. Questi valori sono stati deprecati in quanto non in linea con il principio di riservatezza dei dati interni all'applicazione.

Una volta ottenuto uno Stream, in input o in output che sia, va gestito come normale classe Java per procedere alle operazioni, rispettivamente, di lettura o scrittura.

Storage esterno

Anche per lo Storage esterno, le operazioni su filesystem si svolgono mediante Stream e le consuete classi Java. L'accesso avverrà mediante la classe **Environment**.

La prima operazione da svolgere è controllare lo stato del supporto. Lo si fa con il metodo statico `String Environment.getExternalStorageSta`

La stringa restituita può avere una molteplicità di valori, tutti associati a costanti della classe Environment. Evitando di elencarli tutti, teniamo presente solo che due valori in particolare ci dicono che il supporto può

essere usato:

- *Environment.MEDIA_*
  il caso migliore. Supporto disponibile in lettura/scrittura;
- *Environment.MEDIA_*
  il supporto è disponibile ma solo in lettura.

Tutti gli altri valori indicano situazioni problematiche da valutare a seconda delle circostanze.

Una volta controllato lo stato del supporto e del relativo filesystem, è arrivato il momento di lavorarci direttamente. L'accesso alla cartella

root dello Storage esterno primario si ottiene con il metodo statico: `File getExternalStorageDirectory()`.

Dal riferimento all'oggetto File ottenuto, è possibile, secondo le procedure Java, leggere i contenuti, lavorare sui dati e via dicendo.

È sconsigliabile salvare file direttamente nella cartella principale dello storage esterno, normalmente esso contiene delle cartelle associate alle principali tipologie di contenuti:

- Alarms, per i suoni da abbinare agli allarmi
- Download, per i file

scaricati;

- Movie, per i film;
- Music, per i file musicali;
- Notifications, per i suoni delle notifiche;
- Pictures, per le foto;
- Podcasts, per i file di podcast;
- Ringtones, per le suonerie.

Ciò non impedisce ovviamente che ve ne possano essere altre.

Quella della gestione dei file è il primo caso di persistenza che

incontriamo in questa guida. Eppure per il programmatore riveste una grande importanza, soprattutto perchè serve a gestire foto, musica, suonerie e tanti altri dati dalla forte connotazione personale destinati ad intrecciarsi indissolubilmente con la vita dell'utente-tipo.

Capitolo 28 – Memorizzare informazioni con SharedPreferences

L'utilizzo dei file come collocazione di dati persistenti è molto duttile. La possibilità di salvare dati grezzi mediante Stream apre la strada a moltissime possibilità. Si possono salvare file binari, con codifiche proprie, serializzare strutture dati Java o ricorrere a formati "a caratteri" come CSV o altro ancora.

A volte però, si ha solo bisogno di salvare dati in locale, magari di tipo primitivo, come password, indirizzi IP, numeri o altre informazioni di configurazione piuttosto elementari. In

questi casi più che confrontarsi con la varietà offerta dai file farebbe comodo una specie di mappa in cui salvare coppie chiave/valore con la possibilità di renderla persistente su disco.

Tutto ciò è disponibile e prende il nome di **SharedPreferences**.

I dati collocati nelle SharedPreferences vengono salvati in un file XML contenuto nello storage interno, precisamente nella cartella *shared_prefs*. Il file in cui sono contenute può avere un nome di default o assegnato dal programmatore pertanto di potrà accedere alle Preferences in due modi:

- nella modalità di default con il metodo `getPreferences()`. Questo metodo è concepito per salvare dati privati dell'Activity. Il nome di default prodotto per il file richiamerà infatti il nome dell'Activity;
- specificando un nome di file con `getSharedPreferences filename, int mode)` dove il primo parametro indica il nome che si vuole dare al file di preferenze mentre il

secondo sono i priviligi da concedere al file.

Entrambi i metodi restituiranno un oggetto SharedPreferences sul quale si potrà agire come su di una mappa.

Si noterà subito che un oggetto SharedPreferences contiene tipici metodi di lettura come:

- `contains(String key)` che verifica se una proprietà con una certa etichetta è già stata salvata;
- una serie di metodi *getter* come `getInt`,

`getString`, `getFloat` e via dicendo con i quali si potrà recuperare i dati salvati per una certa chiave.

Tra gli altri, si vedrà anche un metodo `edit()` che restituisce un oggetto di tipo Editor. Questa classe è il **meccanismo di modifica delle SharedPreferences**.

Dall'Editor si avrà accesso a molti metodi put che permettono di modificare le proprietà. Al termine delle modifiche è molto importante che si richieda il salvataggio delle stesse invocando il metodo `apply()`, anch'esso dell'oggetto Editor.

Semplicità delle mappe, utilità della

persistenza, se vi si aggiunge la frequenza in cui si necessita di salvare dati così semplici si comprende perchè le SharedPreferences siano considerate uno strumento importantissimo per il programmatore.

Capitolo 29 – Database e SQLite

Il salvataggio di dati su file – visto nei capitoli precedenti – potrebbe essere sufficiente in molti casi. In fin dei conti le API di I/O fornite dal linguaggio Java permettono di trattare dati binari e testuali, salvare strutture dati serializzate ed altro ancora.

Qualcosa però a cui il programmatore è particolarmente abituato è il database relazionale e l'interazione mediante linguaggio SQL tanto da sentirne il bisogno anche in Android. L'evidenza di soddisfare questa necessità richiedeva che venisse individuato un prodotto dotato di determinati requisiti: open-

source, ampiamente diffuso, mantenuto e documentato da una comunità prospera, efficiente e soprattutto che non richiedesse l'esecuzione di un servizio continuo in background. La soluzione esisteva già nel mondo del software libero e risiedeva in **SQLite**.

SQLite

SQLite è considerato il motore di database più diffuso al mondo. Rispetta tutti i requisiti di efficienza e disponibilità di cui si è detto. **Si tratta, in realtà, di una libreria software che permette di gestire in un unico file un database relazionale**.

Oltretutto è un progetto in continua espansione che mette a disposizione molti aspetti dei moderni DBMS: View, Trigger, transazioni, indici oltre al comunissimo e comodissimo interfacciamento con linguaggio SQL.

Nota: per chi non lo sapesse, SQL è il linguaggio per inviare comandi ad un database ed estrapolarne dati. È il

formalismo con cui vengono realizzate le ben note *query*. In questa sede non ci si dilungherà sull'argomento ma se ne darà per assodata la conoscenza da parte del lettore. Qualora così non fosse, si ritiene opportuno un approfondimento su specifica documentazione.

Database nelle proprie App

Per avere un database SQLite nella propria App Android, non è necessario scaricare né installare niente: semplicemente basta chiedere. La libreria SQLite infatti è già inclusa nel sistema operativo e le API disponibili nel framework offrono tutto il supporto necessario.

Questi i passi:

- **creare la struttura del database**. Il programmatore dovrà preparare uno script SQL

che crei la struttura interna del database (tabelle, viste ecc.). Nel realizzarla, potrà procedere nella maniera che gli è più congeniale scrivendola a mano o aiutandosi con uno strumento visuale come Sqliteman. L'importante è che alla fine di questa fase abbia una stringa contenente i comandi di creazione;

- **creare una classe Java che estenda** `SQLiteOpenHelper`. Questa classe, che nel

seguito chiameremo *helper*, servirà a gestire la nascita e l'aggiornamento del database su memoria fisica e a recuperare un riferimento all'oggetto SQLiteDatabase, usato come accesso ai dati;

- **creare una classe per l'interazione con il database**. Solitamente questa ha due caratteristiche: (1) contiene un riferimento all'oggetto *helper* definito al punto precedente, (2) contiene i

metodi con cui, dalle altre componenti dell'app, verranno richieste operazioni e selezioni sui dati.

Puntualizziamo che i tre step appena enunciati non sono assolutamente obbligatori, esistono infatti modalità alternative di azione. Sono tuttavia una prassi molto comune e funzionale per l'approntamento di un database a supporto di un'app. Tanto verrà dimostrato con l'esempio a seguire. Se ne consiglia perciò l'osservanza.

Esempio pratico

Verrà creata un'Activity che gestisce un piccolo scadenziario. I dati inseriti saranno costituiti da un oggetto, un testo che costituisce il vero promemoria ed una data.

Mettiamo subito in pratica i primi due step: creazione del database e della classe *helper*.

```
public class DBhelper extends SQLiteOpenHelper
{
        public static final String DBNAME="BILLBOOK";

    public DBhelper(Context context) {
        super(context, DBNAME, null, 1);
```

```
        }


        @Override
         public void onCreate(SQLiteDatabase db)

        {
                        String q="CREATE TABLE
"+DatabaseStrings.TBL_NAME+
                         " ( _id INTEGER PRIMARY
KEY AUTOINCREMENT," +
                        DatabaseStrings.FIELD_SU
TEXT," +
                        DatabaseStrings.FIELD_TE
TEXT," +
                        DatabaseStrings.FIELD_DA
TEXT)";
                db.execSQL(q);
        }
```

```
        @Override
            public void onUpgrade(SQLiteDatabase
db, int oldVersion, int newVersion)
        { }


    }
```

Per questioni “organizzative” del codice, i nomi dei campi e della tabella sono stati definiti in costanti nella seguente classe:

```
    public class DatabaseStrings
    {
        public static final String FIELD_ID="_id";
                    public static final String
FIELD_SUBJECT="oggetto";
```

public static final String FIELD_TEXT="testo";

public static final String FIELD_DATE="data";

public static final String TBL_NAME="Scadenze";

}

Notiamo che per prima cosa viene creato un **costruttore** al cui interno si invoca quello della classe base:

```
super(context, DBNAME, null, 1);
```

Tra gli argomenti passati ne notiamo due in particolare:

- il **nome del database**: è il secondo parametro,

di tipo String, valorizzato con la costante DBNAME. Questo è il nome che il database avrà nello spazio disco dell'applicazione;

- la **versione del database**: è il quarto argomento, di tipo intero e valore 1.

Inoltre è stato fatto l'override di due metodi:

- `onCreate`: viene invocato nel momento in cui non si trova nello

spazio dell'applicazione un database con nome indicato nel costruttore. Da ricordare che onCreate verrà invocato una sola volta, quando il database non esiste ancora. Il parametro passato in input è un riferimento all'oggetto che astrae il database. **La classe SQLiteDatabase è importantissima in quanto per suo tramite invieremo i comandi di gestione dei dati**. Il metodo onCreate contiene la query SQL

che serve a creare il contenuto del database. Questo è l'applicazione del primo step, enunciato prima. Notare che al suo interno non c'è alcun comando `CREATE DATABASE` in quanto il database stesso è già stato creato dal sistema. Il comando SQL di creazione verrà invocato mediante `execSQL`;

- `onUpgrade`: viene invocato nel momento in cui si richiede una versione del database più aggiornata di quella

presente su disco. Questo metodo contiene solitamente alcune query che permettono di adeguare il database alla versione richiesta.

La classe in cui gestiremo il database prende il nome di `DbManager`, ne vediamo subito il codice:

```
public class DbManager
{
    private DBhelper dbhelper;

    public DbManager(Context ctx)
    {
        dbhelper=new DBhelper(ctx);
```

```
        }


            public void save(String sub, String txt,
String date)
        {
                                        SQLiteDatabase
db=dbhelper.getWritableDatabase();


                                    ContentValues cv=new
ContentValues();
            cv.put(DatabaseStrings.FIELD_SUBJ
sub);
            cv.put(DatabaseStrings.FIELD_TEXT,
txt);
            cv.put(DatabaseStrings.FIELD_DATE,
date);
            try
            {
```

```
                    db.insert(DatabaseStrings.TBL_N
null,cv);
                }
                catch (SQLiteException sqle)
                {
                    // Gestione delle eccezioni
                }
            }

            public boolean delete(long id)
            {
                                        SQLiteDatabase
db=dbhelper.getWritableDatabase();
                try
                {
                                                    if
(db.delete(DatabaseStrings.TBL_NAME,
DatabaseStrings.FIELD_ID+"=?",   new   String[]
```

```
{Long.toString(id)})>0)
                        return true;
                    return false;
                }
                catch (SQLiteException sqle)
                {
                    return false;
                }

        }

        public Cursor query()
        {
            Cursor crs=null;
            try
            {
```

```
                                        SQLiteDatabase
db=dbhelper.getReadableDatabase();
                        crs=db.query(DatabaseStrings.T
null, null, null, null, null, null, null);
                }
                catch(SQLiteException sqle)
                {
                        return null;
                }
                return crs;
        }

    }
```

Prima cosa da notare: **la classe contiene un riferimento al DbHelper**.

I metodi che vengono implementati

mostrano tre operazioni basilari da svolgere sulla tabella del db: *save* per salvare una nuova scadenza, *delete* per cancellarne una in base all'id, *query* per recuperarne l'intero contenuto.

Da questi metodi, **emerge un** *modus operandi* **comune**. Infatti per lavorare su un oggetto SQLiteDatabase, la prima cosa da fare è recuperarne un riferimento. Lo si può fare con i metodi di SQLiteOpenHelper, `getReadableDatabase()` e `getWriteableDatabase()` che restituiscono, rispettivamente, un riferimento al database "in sola lettura" e uno che ne permette la modifica.

Sull'oggetto SQliteDatabase

recuperato, si svolge una delle quattro operazioni CRUD, le azioni fondamentali della persistenza (*Create, Read, Update, Delete*).

Nelle API Android per Sqlite esiste almeno un metodo per ogni tipo di azione:

- `query`: esegue la lettura sulle tabelle: mette in pratica il SELECT sui dati. I suoi svariati overload predispongono argomenti per ogni parametro che può essere inserito in una interrogazione di questo

tipo (selezione, ordinamento, numero massimo di record, raggruppamento, etc.);

- `delete`: per la cancellazione di uno o più record della tabella;
- `insert`: per l'inserimento. Riceve in input una stringa che contiene il nome della tabella e la lista di valori di inizializzazione del nuovo record mediante la classe **ContentValues**. Questa è una struttura a mappa che accetta coppie chiave/valore dove la

chiave rappresenta il nome del campo della tabella;

- `update`: esegue modifiche. Il metodo associa i parametri usati nell'insert e nel delete.

Tutti questi metodi non richiedono un uso esplicito di SQL. Chi ne avesse bisogno o preferisse per altre ragioni scrivere totalmente i propri comandi e query può utilizzare metodi di SqliteDatabase come `execSQL` e `rawQuery`.

Vale anche la pena sottolineare che i metodi appena indicati offrono una versione "parametrica" delle condizioni

di selezione dei record (la classica clausola WHERE di SQL che spesso è indispensabile in selezioni, cancellazioni e aggiornamenti). Ciò è visibile nella classe DbManager, nel metodo che si occupa della cancellazione:

db.delete(DatabaseStrings.TBL_NAME, DatabaseStrings.FIELD_ID+"=?", new String[] {Long.toString(id)})>0

In questi casi, la classe SQLiteDatabase vuole che una stringa raccolga la parte fissa del contenuto della clausola `WHERE` sostituendo le parti variabili con punti interrogativi. Gli argomenti attuali verranno passati ad ogni invocazione in un array di stringhe.

Nell'esecuzione della query ogni punto interrogativo verrà, in ordine, sostituito con un parametro dell'array.

Altra classe cui fare attenzione, è **Cursor**. Rappresenta un puntatore ad un set di risultati della query. Somiglia a quell'elemento che in altre tecnologie prende il nome di `RecordSet` o `ResultSet`. Un oggetto Cursor può essere spostato per puntare ad una riga differente del set di risultati. Ciò viene fatto con i metodi `moveToNext`, `moveToFirst`, `moveToLast` e così via.

Una volta che il cursore ha raggiunto la riga desiderata si può passare alla lettura dei dati con metodi specifici in base al tipo di dato (`getString`, `getLong` ecc.) indicando il nome del

campo.

Ad esempio, se l'oggetto *crs* di classe Cursor punta ad un insieme di righe della tabella Scadenze, una volta indirizzato sulla riga desiderata si potrà leggere il campo relativo all'oggetto con:

crs.getString(crs.getColumnIndex(DatabaseStrin

Con `getColumnIndex` viene trovato l'indice del campo.

L’Activity ed il CursorAdapter

L’interfaccia utente che si occuperà di interagire con il db è molto semplice.


Oggetto:
saldo lavoro
Testo:
importo 250 euro
Data:
12/06/2014
Salva
pagare bolletta
12/03/2014
pagare acconto
12/05/2014

Costituita da un form per l'inserimento di nuove scadenze e da una ListView sottostante che mostra i record presenti nel db, permette tuttavia di sperimentare le funzionalità sinora descritte.

```
public class MainActivity extends Activity
{

    private DbManager db=null;
    private CursorAdapter adapter;
    private ListView listview=null;
    private OnClickListener clickListener=new
View.OnClickListener()
    {
        @Override
        public void onClick(View v)
```

```
            {
                int position=listview.getPositionForView(v);
                long id=adapter.getItemId(position);
                    if (db.delete(id))
                        adapter.changeCursor(db.q
            }
        };
        @Override
            protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main
            db=new DbManager(this);
                listview=(ListView) findViewById(R.id.listview);
            Cursor crs=db.query();
```

```
adapter=new CursorAdapter(this,
crs, 0)
{
@Override
public View newView(Context
ctx, Cursor arg1, ViewGroup arg2)
{
View
v=getLayoutInflater().inflate(R.layout.listactivity_rc
null);
return v;
}
@Override
public void bindView(View v,
Context arg1, Cursor crs)
{
String
oggetto=crs.getString(crs.getColumnIndex(Databas
```

```
                                        String
data=crs.getString(crs.getColumnIndex(DatabaseSt

                                    TextView txt=(TextView)
v.findViewById(R.id.txt_subject);

                                    txt.setText(oggetto);

                                            txt=(TextView)
v.findViewById(R.id.txt_date);

                                    txt.setText(data);

                                        ImageButton imgbtn=
(ImageButton) v.findViewById(R.id.btn_delete);

                                    imgbtn.setOnClickListener(c

                        }


                        @Override

                            public long getItemId(int
position)

                        {

                                            Cursor
```

```
crs=adapter.getCursor();
                    crs.moveToPosition(position
                                        return
crs.getLong(crs.getColumnIndex(DatabaseStrings.I
                }
            };

            listview.setAdapter(adapter);
        }

        public void salva(View v)
        {
                        EditText  sub=(EditText)
findViewById(R.id.oggetto);
                         EditText  txt=(EditText)
findViewById(R.id.testo);
                        EditText  date=(EditText)
findViewById(R.id.data);
```

```
                    if (sub.length()>0 &&
date.length()>0)
                    {
                        db.save(sub.getEditableText().toS
txt.getEditableText().toString(),
date.getEditableText().toString());
                        adapter.changeCursor(db.query(
                    }
                }
            }
```

L'Activity gestisce l'interazione con il database appellandosi all'oggetto DbManager istanziato. Il metodo *salva* viene invocato al click del pulsante del form mentre l'oggetto `OnClickListener` serve ad ogni

pulsante di cancellazione presente sulle righe della ListView.

Fin qui niente di particolarmente sorprendente. L'elemento di maggiore novità è l'Adapter che è stato usato: il **CursorAdapter**. Il suo scopo è trasformare ogni riga del risultato della query in una View.

Nell'esempio, il layout usato per mostrare la singola riga è il seguente:

```
<RelativeLayout
    android:layout_height="wrap_content"
    android:layout_width="400dp"
    xmlns:android="http://schemas.android.co
  <LinearLayout
    android:layout_height="wrap_content"
```

```
    android:layout_width="match_parent"
    android:padding="5dp"
    android:layout_toLeftOf="@+id/btn_delet
    android:orientation="vertical">
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="@style/big_textstyle"
    android:id="@+id/txt_subject"/>
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="@style/small_textstyle"
    android:id="@+id/txt_date"/>
</LinearLayout>
<ImageButton
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
        android:src="@android:drawable/ic_men
        android:id="@+id/btn_delete"
        />
</RelativeLayout>
```

Il CursorAdapter lo tratterà mediante override di due metodi in particolare:

- `newView`: crea la View cui associare i dati prelevati dal database. Ciò viene fatto, in questo

e molti altri casi, mediante *inflating*;

- `bindView`: riceve in input una View, da completare con i dati di un singolo record del cursore, e il Cursor già posizionato sulla riga giusta. Grazie all'implementazione di newView, questo metodo riceverà sempre una View pronta, creata per l'occasione o "riciclata" da quelle già esistenti. Le operazioni dominanti in bindView riguarderanno il recupero di controlli

presenti nella View ed il loro completamento con i dati presi dal Cursor.

Ciò che viene fatto all'interno dei predetti metodi non dovrebbe stupire più in quanto sono le stesse operazioni fatte per gli Adapter customizzati già presentati in questa guida.

Un terzo metodo frutto di override nel CursorAdapter è `getItemId`. Fornisce l'id del record in base alla posizione e viene usato per completare le condizioni di selezione richieste per la cancellazione.

Capitolo 30 – Condividere i dati con i Content Provider

Uno dei principi che è più volte riemerso nel corso di questa guida è la riservatezza dei dati dell'applicazione. Lo si è anticipato già nei primi capitoli e lo si è sperimentato studiando i metodi di persistenza: i file ed i database creati devono essere utilizzati solo dall'applicazione cui appartengono. In effetti, l'esistenza dello stesso Internal Storage ne è già una conferma.

Ma allora – ci si potrebbe chiedere – come possono le applicazioni condividere i propri dati con il resto del sistema?

Un meccanismo di condivisione esiste ed è rappresentato dai **ContentProvider,** una delle quattro componenti delle app Android, oltre ad Activity, Service e BroadcastReceiver.

Questo capitolo si occuperà di mostrare le caratteristiche di base di un ContentProvider e di come sia possibile crearne uno nella propria applicazione. L'esempio che ci guiderà in questo percorso sarà la "trasformazione" in ContentProvider del database a supporto dello scadenziario visto nel capitolo precedente.

Funzionamento di base di un ContentProvider

I ContentProvider sono riconoscibili mediante un **URI**, un riferimento univoco. Questi indirizzi sono costituiti da due parti:

- *authority*: è il nome del Provider in generale. Spesso, per evitare conflitti, richiama il nome del package Java di appartenenza;
- *path*: costituisce un percorso interno alla singola authority. Spesso

rappresenta una tabella del database cui si vuole accedere ma, in generale, indica la tipologia di dati su cui agire.

Un ContentProvider, come avviene per le Activity, deve essere definito in *due step*:

- creare una classe Java che estenda ContentProvider;
- creare un nodo <provider> all'interno dell'AndroidManifest.xml Servirà ad associare la

classe Java che implementa il Provider (definita al punto precedente) e l'authority scelta per gli URI.

Appena definita la classe ContentProvider, viene richiesto di implementarne i metodi astratti, tra cui query, update, delete e insert. Definiscono le operazioni CRUD ed i parametri richiesti ricordano in tutto – tipologia e funzioni – quelli usati per i database come illustrato nel capitolo precedente. La differenza sarà la mancanza del nome della tabella, sostituita dall'indicazione dell'URI.

Svolti questi passi, qualunque

applicazione nel sistema sarà in grado di accedere alla sorgente dati sottesa al ContentProvider.

Sarà sufficiente fare accesso ad un oggetto di sistema, detto **ContentResolver** e chiedergli di svolgere operazioni sugli URI conosciuti del ContentProvider. **Ogni operazione CRUD che verrà chiesta al ContentResolver sarà convertita in una chiamata al corrispondente metodo CRUD della classe ContentProvider**.

Il ContentResolver sarà in grado di stabilire la relazione tra URI e classe Java grazie ai dati registrati nel manifest dell'applicazione.

Adattamento del database a ContentProvider

Per attuare l'adattamento del database a ContentProvider, iniziamo con la definizione degli URI. Scegliamo il package Java ("it.html.guida.database") come authority e che tutti i path inizino con il segmento "/scadenze". Queste sono decisioni che spettano al programmatore.

Le azioni che potranno essere invocate sul ContentProvider sono le stesse del precedente capitolo:

- l'elenco delle scadenze: una query che

recupera tutto il contenuto della tabella;

- inserire una nuova scadenza;
- richiede la cancellazione di una scadenza riconosciuta mediante l'id;

e questi gli URI presso i quali potranno, rispettivamente, essere richieste:

- content:// it.html.guida.database/sca

- content://

it.html.guida.database/sca

- content://
  it.html.guida.database/sca

Il ContentProvider verrà implementato dalla classe BillBookProvider mentre la MainActivity rimarrà praticamente invariata. Non sarà più necessario usare oggetti DbManager per accedere ai dati. Il collegamento alla sorgente dati avverrà, in maniera “remota”, attraverso URI e ContentResolver.

All’atto pratico questo determinerà, innanzitutto, che il collegamento al helper sarà contenuto nella classe

ContentProvider ed è qui che queste componenti dimostrano un aspetto molto importante in termini architetturali: il disaccoppiamento totale tra lo strato di presentazione (l'Activity) e il livello di accesso ai dati.

La registrazione del ContentProvider nell'AndroidManifest.xml avverrà così:

```
<application
. . .
. . .>
    <activity
    . . .
    . . ./>
<provider  android:name=".BillBookProvider"
```

```
android:authorities="it.html.guida.database"/>
    </application>
```

associando authority e classe Java.

Ecco il ContentProvider:

```
    public class BillBookProvider extends
ContentProvider
    {

        private DBhelper dbhelper=null;

        @Override
        public boolean onCreate()
        {
```

```
        bhelper=new DBhelper(getContext());

        return true;

    }


    @Override

        public Cursor query(Uri uri, String[]
projection, String selection,String[] selectionArgs,
String sortOrder)

    {

        Cursor crs=null;

        try

        {

                                        SQLiteDatabase
db=dbhelper.getReadableDatabase();

            crs=db.query(DatabaseStrings.TBL_NAM
null, null, null, null, null, null, null);
```

```
        }
          catch(SQLiteException sqle)
          {
              return null;
          }
          return crs;
        }

        @Override
          public int delete(Uri uri, String selection,
String[] selectionArgs)
        {
                                          SQLiteDatabase
db=dbhelper.getWritableDatabase();
```

```
        int res=-1;
        String id=uri.getLastPathSegment();
        try
                {
            res=db.delete(DatabaseStrings.TBL_NAM
DatabaseStrings.FIELD_ID+"=?",  new  String[]
{id});
        }
        catch (SQLiteException sqle)
        {
            // Gestione delle eccezioni
        }
        return res;
    }
```

```
@Override
    public Uri insert(Uri uri, ContentValues values)
    {
                                        SQLiteDatabase db=dbhelper.getWritableDatabase();
        long id=-1;
        try
        {
            db.insert(DatabaseStrings.TBL_NAME, null,values);
        }
        catch (SQLiteException sqle)
        { return null; }
                return  Uri.withAppendedPath(uri,
```

```
Long.toString(id));
        }

        @Override
            public int update(Uri uri, ContentValues
values, String selection,String[] selectionArgs)
        {   // modifica non implementata
            return 0; }

        @Override
        public String getType(Uri uri)
        { return null; }

    }
```

Non ripetiamo il codice dell'Activity perchè ha subito, come detto, modifiche minime. È interessante però notare come cambia l'accesso alla persistenza.

Senza ContentProvider, avremmo chiesto l'elenco delle scadenze così:

```
db=new DbManager(this);

Cursor crs=db.query();
```

ora, non abbiamo più bisogno del DbManager, pertanto faremo questo:

```
Cursor crs=getContentResolver().query(U
content://it.html.guida.database/scadenza
null, null, null, null);
```

Il riferimento al ContentResolver viene fornito dal Context. Analogamente potremo richiedere l'inserimento di una nuova scadenza all'interno del metodo *salva*:

```
ContentValues cv=new ContentValues();

cv.put(DatabaseStrings.FIELD_SUBJECT,
sub.getEditableText().toString());
            cv.put(DatabaseStrings.FIELD_TEXT,
txt.getEditableText().toString());
            cv.put(DatabaseStrings.FIELD_DATE,
date.getEditableText().toString());
getContentResolver().insert(Uri.parse("conte
/scadenze/nuova"), cv);
```

o la cancellazione mediante id

nell'implementazione di OnClickListener:

```
long id=adapter.getItemId(position); // id
dell'elemento

getContentResolver().delete(Uri.withAppende
("content://it.html.guida.databse/scadenze/elimin
Long.toString(id)),null,null);
```

Ultima nota, lavorando con gli URI, è utile guardare con attenzione la documentazione per scoprire i vari metodi di utilità che si possono usare. Ne sono esempio *parse* e *withAppendedPath* qui utilizzati.

Capitolo 31 – Accedere a ContentProvider

I dispositivi Android gestiscono molti dati come contatti, file archiviati su disco, eventi del Calendario. Tutte queste informazioni possono essere lette e modificate dalle nostre applicazioni mediante ContentProvider. Il carattere particolarmente "personale" di questi dati segnala quanto la programmazione di un dispositivo mobile a volte si allontani dall'astrazione dell'informatica e si intrecci fortemente con la vita reale dell'utente.

Nel capitolo precedente abbiamo visto le caratteristiche dei ContentProvider, funzionamento e

validità architetturale. Qui apprezzeremo molto la standardizzazione dei meccanismi di accesso che mettono a disposizione, agevolando la consultazione di basi di dati spesso molto diverse tra loro.

Aspetti cui si dovrà porgere particolare attenzione:

- **gli URI non sono più inventati dal programmatore** ma questa volta sono parte delle API di accesso al ContentProvider. Il framework cercherà tuttavia di renderne

semplice la conoscenza mediante apposite classi che prendono il nome di **classi Contract**. Essenzialmente si tratta di classi che contengono un gran numero di costanti, generalmente suddivise in sottoclassi, che forniscono i nomi delle tabelle dei dati, dei campi ed altro ancora sulla struttura della sorgente dati;

- solitamente sarà necessario aggiungere delle **permission** all'interno del file

AndroidManifest.xml. Saranno probabilmente più di una, relative a lettura dei dati e alla scrittura.

Per quanto riguarda l'accesso, si userà ugualmente il ContentResolver ed i suoi metodi che permetteranno di mettere in pratica le quattro operazioni CRUD: lettura, inserimento, modifica e cancellazione.

Alcuni ContentProvider di sistema

ContentProvider molto noti del sistema opearativo sono:

- **Contacts**: include tutte le informazioni sui contatti dell'utente: rubrica telefonica, email, etc. Non se ne parlerà in questo capitolo. Troverà spazio in un contesto più adeguato come l'esplorazione del rapporto tra Android e telefonia;

- **MediaStore**: gestisce dati relativi a file multimediali contenuti nel sistema tra cui file audio, immagini e video;

- **UserDictionary**: si occupa delle parole aggiunte dall'utente al dizionario di default;

- **Calendar**: basato su API disponibili da Android 4 (IceCreamSandwich) e serve a gestire appuntamenti ed eventi sul calendario del

dispositivo eventualmente sincronizzato con l’account Google. Verrà illustrato nel prossimo paragrafo.

Un esempio: gestire il Calendario

Le API Calendar sono state aggiunte in Android 4 e possono essere davvero utili per fare in modo che le nostre applicazioni possano aiutare gli utenti a gestire i propri impegni.

Per utilizzare questo provider è necessario innanzitutto aggiungere al manifest le permission richieste, dipendentemente dal tipo di operazioni che si vogliono svolgere (lettura e/o scrittura):

<uses-permission

android:name="android.permission.REAL

<uses-permission

*android:name="android.permission.WRITE_*

Tutto il sistema ad oggetti del calendario verterà intorno alla classe Contract di competenza, ovviamente CalendarContract. Questa permetterà l'accesso ad una serie di tabelle, ognuna dedicata ad un aspetto. Ecco le più comuni:

- `CalendarContract.Ca` contiene le informazioni su ogni Calendario. Nel sistema infatti possono essere disponibili più calendari, alcuni di carattere locale quindi

utilizzabili solo sul dispositivo, altri legati ad un account Google. Quest'ultimo caso permetterà una sincronizzazione tra i dati presenti nel telefono e quelli remoti;

- `CalendarContract.Ev` è una delle tabelle più importanti ed elenca gli eventi memorizzati nei calendari. L'id del singolo evento servirà per stabilire un collegamento tra questa tabella e altre che ne specificano alcune

sfaccettature come Reminders e Attendees;

- `CalendarContract.At` è collegata a Events e memorizza i partecipanti all'evento specificandone, tra l'altro, il nome e l'account;
- `CalendarContract.Re` collegata anch'essa a Events, ogni sua riga simboleggia un alert o una notifica impostata per un evento. Ogni evento può avere anche più notifiche.

Vediamo ora alcuni esempi di codice per interagire con il nostro calendario.

Recuperare i calendari disponibili nel sistema:

```
String[] projection =
        new String[]{
            Calendars._ID,
            Calendars.NAME,
            Calendars.ACCOUNT_NAME,
            Calendars.ACCOUNT_TYPE};
    Cursor cursor =
        getContentResolver().
            query(Calendars.CONTENT_URI
                projection,
                    Calendars.VISIBLE + " =
```

1",

null,

null);

Per vedere tutti i calendari che possono essere trovati nel sistema, come presumibile, è necessario inviare una query mediante ContentResolver. Si faccia caso che l'URI da utilizzare sarà fornito direttamente dalla classe Calendars e così anche per quanto riguarda i campi. Per il resto non c'è niente di nuovo. Il risultato sarà un normale Cursor che si potrà gestire nelle modalità consuete, lettura diretta o CursorAdapter per citarne alcuni.

Inserire un evento in un determinato calendario:

Una volta utilizzato il codice precedente abbiamo a disposizione tutti i calendari registrati nel dispositivo. Come si può vedere osservando l'array projection, tra i campi recuperati dalla query c'è l'ID. A questo si potrà inserire un nuovo evento nel sistema collegandolo ad un calendario fornendone l'ID.

```
Calendar cal = new GregorianCalendar(2014, 4, 20);

        cal.setTimeZone(TimeZone.getDefault());

        cal.set(Calendar.HOUR, 15);

        cal.set(Calendar.MINUTE, 30);
```

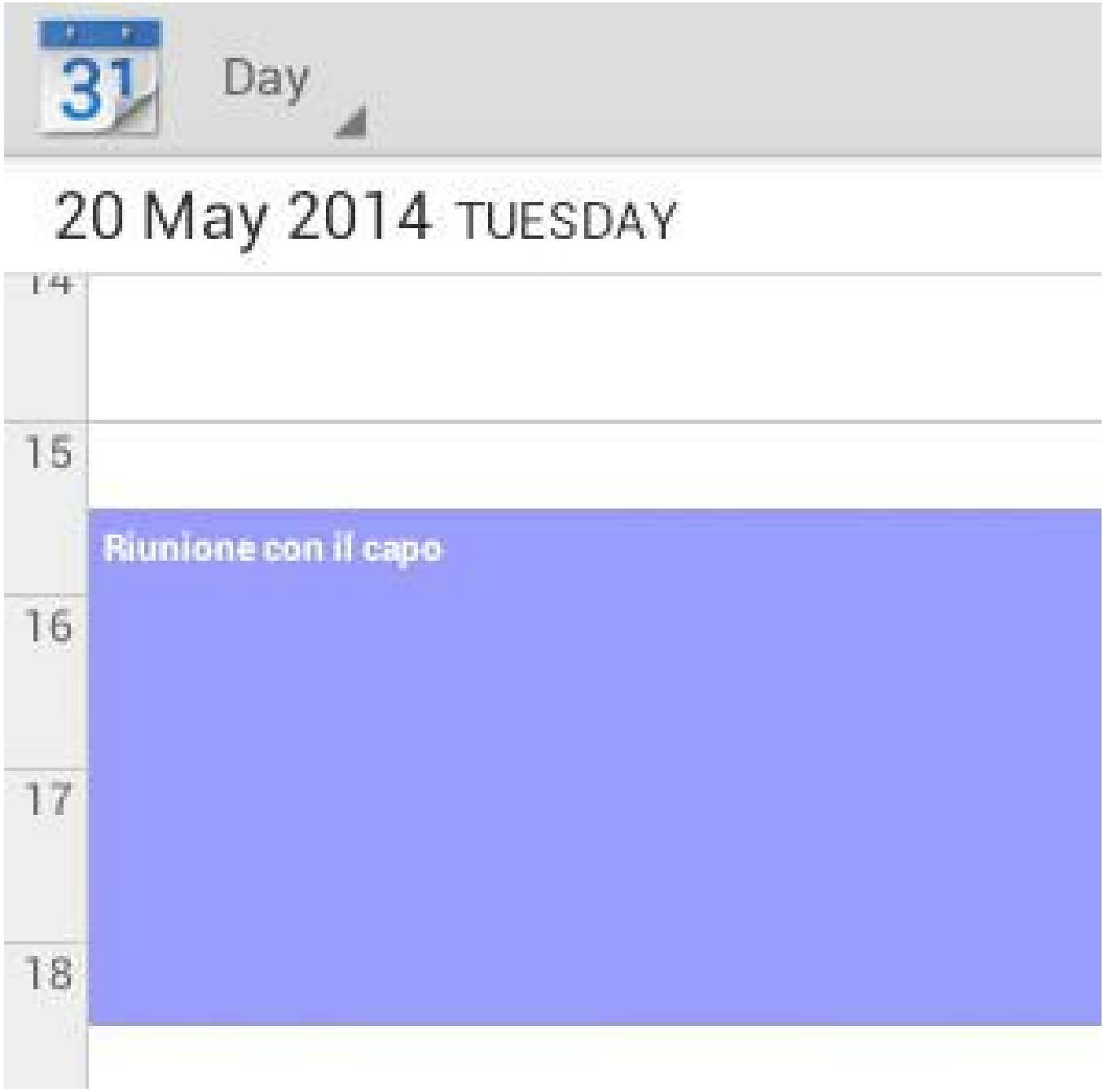
```
long dtstart = cal.getTimeInMillis();
ContentValues values = new ContentValues();
values.put(Events.DTSTART, dtstart);
values.put(Events.DTEND, dtstart+3*3600*1000); // durata di tre ore
values.put(Events.TITLE, "Riunione con il capo");
values.put(Events.CALENDAR_ID, id);
values.put(Events.EVENT_TIMEZONE, TimeZone.getDefault().getDisplayName());
Uri uri =
getContentResolver().
insert(Events.CONTENT_URI, values);
```

Nell'esempio abbiamo salvato un promemoria per la riunione con il capo per il 20 maggio. Abbiamo annotato anche che l'impegno durerà tre ore a partire dalle 15:30.

Il salvataggio effettuato si preoccuperà di inserire solo i dati minimi indispensabili ossia quelli assolutamente obbligatori per un evento non ripetitivo. Tra gli altri vediamo che nel `ContentValues` preparato per l'inserimento è stato collocato anche l'ID del calendario, informazione che può essere recuperata mediante il primo esempio di codice.

Aprendo successivamente l'applicazione Calendar sul dispositivo Android vedremo che effettivamente il

promemoria per la riunione è stato salvato ed è visibile nell'immagine seguente.



Capitolo 32 – Threading

Quando si affronta in maniera approfondita lo studio di una **tecnologia di programmazione** si sa che prima o poi si finisce col trattare un argomento piuttosto spinoso: i thread ed il loro impiego nella programmazione concorrente.

I thread costituiscono un **filone di esecuzione interno ad un'applicazione**. In pratica, sono la vera anima operativa di un programma a runtime. Solitamente un qualunque software – nel mondo mobile e non – vede contemporaneamente attivi più thread. Ognuno di essi svolgerà in maniera a sé stante una sequenza di operazioni, come

se fosse “un programma nel programma”.

Che ruolo giocano i thread quindi nelle app Android e perchè dovremmo tenerli in considerazione? Si consideri che quando si utilizza un’Activity il thread principale dell’applicazione si occupa prevalentemente di gestire i messaggi relativi al funzionamento dell’interfaccia utente. Svolgere in questo stesso thread operazioni presumibilmente “lente” come ad esempio la lettura e scrittura da file, il prelevamento di dati da un database, il caricamento di immagini rischierebbe di rendere poco reattiva la UI. Conseguenza di ciò sarebbe una *user exprerience* non troppo gradevole che

porterebbe l'utente a sostituire la nostra app con altre molto più scattanti.

La reattività dell'interfaccia è uno degli aspetti delle applicazioni più determinanti per il successo presso il pubblico e la longevità sui market.

Operazioni "lente" dovrebbero essere preferibilmente svolte su thread secondari, detti anche *worker thread*. Inoltre l'accesso in Rete, importantissimo nella programmazione moderna ma contraddistinto da tempi di latenza variabili, deve obbligatoriamente essere eseguito su un thread secondario.

I thread come in Java

Tutto ciò che si è appreso nello studio di Java sui thread (estensione della classe Thread, API della concorrenza, Executors) può essere utilizzato in Android.

Ad esempio, se in un'Activity volessimo distaccare alcune attività su un thread secondario sarebbe lecito usare una forma di questo tipo:

```
new Thread()
    {
        @Override
        public void run()
        {
```

```
        /*
          * Inserire QUI il codice
          * da svolgere nel thread secondario
          */
    }
}.start();
```

Ci sono **due problemi** però. Il primo è che il comune uso dei thread non è molto semplice da usare in maniera corretta. Il secondo è che, in Android, da un thread secondario non è possibile modificare l'interfaccia utente senza usare opportuni meccanismi di comunicazione.

Per fortuna il framework offre un'alternativa che permette di usare un

thread secondario in maniera corretta senza il problema di dover gestire la comunicazione tra thread: la classe **AsyncTask**.

AsyncTask

Per illustrare la classe AsyncTask si mostrerà un esempio poco pratico ma dal forte significato concettuale.

Immaginiamo di avere un'Activity che contiene un solo pulsante. Al click di tale controllo viene **attivato un lavoro in background**. In questo caso si tratterà di un lavoro puramente fittizio: a scopo di esempio lasceremo il thread in attesa per alcuni secondi tanto per generare ritardo. A scandire i tempi dell'attività in background, ci sarà una finestra di dialogo di tipo ProgressDialog.

Il layout dell'activity è molto semplice, contiene infatti un solo pulsante collocato in posizione centrale.

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/re
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:onClick="start"
        android:text="@string/btn_start" />
</RelativeLayout>
```

Il codice Java della classe invece è il seguente:

```
public class MainActivity extends Activity
{
    ProgressDialog progress=null;

    @Override
            protected void onCreate(Bundle
savedInstanceState)
    {
        super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
                                    progress=new
ProgressDialog(MainActivity.this);
        progress.setMax(100);
        progress.setMessage(getString(R.string.pro
        progress.setProgressStyle(ProgressDialog.
```

```
        progress.setCancelable(false);


    }


    public void start(View v)
    {
        new BackgroundTask().execute();


    }


        private class BackgroundTask extends
AsyncTask<Void, Integer, String>
    {


        @Override
```

```
protected void onPreExecute() {
    super.onPreExecute();
    progress.setProgress(0);
    progress.show();
}

@Override
  protected String doInBackground(Void...
arg0)
{
    try
    {
        for(int i=0;i<10;i++)
            {
```

```
                    publishProgress(new Integer[]
{i*10});
                        Thread.sleep(1200);
                    }
                }
                catch (InterruptedException e)
                {}

                return "Lavoro Terminato!";
            }

            @Override
                                protected void
onProgressUpdate(Integer... values)
            {
```

```
            super.onProgressUpdate(values);
            progress.setProgress(values[0].intValue(
        }

        @Override
            protected void onPostExecute(String
result)
        {
            super.onPostExecute(result);
            progress.dismiss();
    Toast.makeText(MainActivity.this,          result,
Toast.LENGTH_SHORT).show();
        }

    }
```

```
}
```

Al click del pulsante viene invocato il metodo `start()` al cui interno si istanzia la classe BackgroundTask e la si manda in esecuzione direttamente. La classe **BackgroundTask estende AsyncTask** e pertanto rappresenta il cuore dell'esempio.

I metodi contenuti nella classe BackgroundTask sono di due tipi diversi. **Il metodo doInBackground è l'unico di quelli implementati che viene eseguito su un thread secondario**. Qui dovremo collocare tutte le operazioni "lente".

Al contrario, `onPreExecute`,

onPostExecute e **onProgressUpdate** sono eseguiti sul thread principale e **si occupano della comunicazione tra thread**.

Rispettivamente:

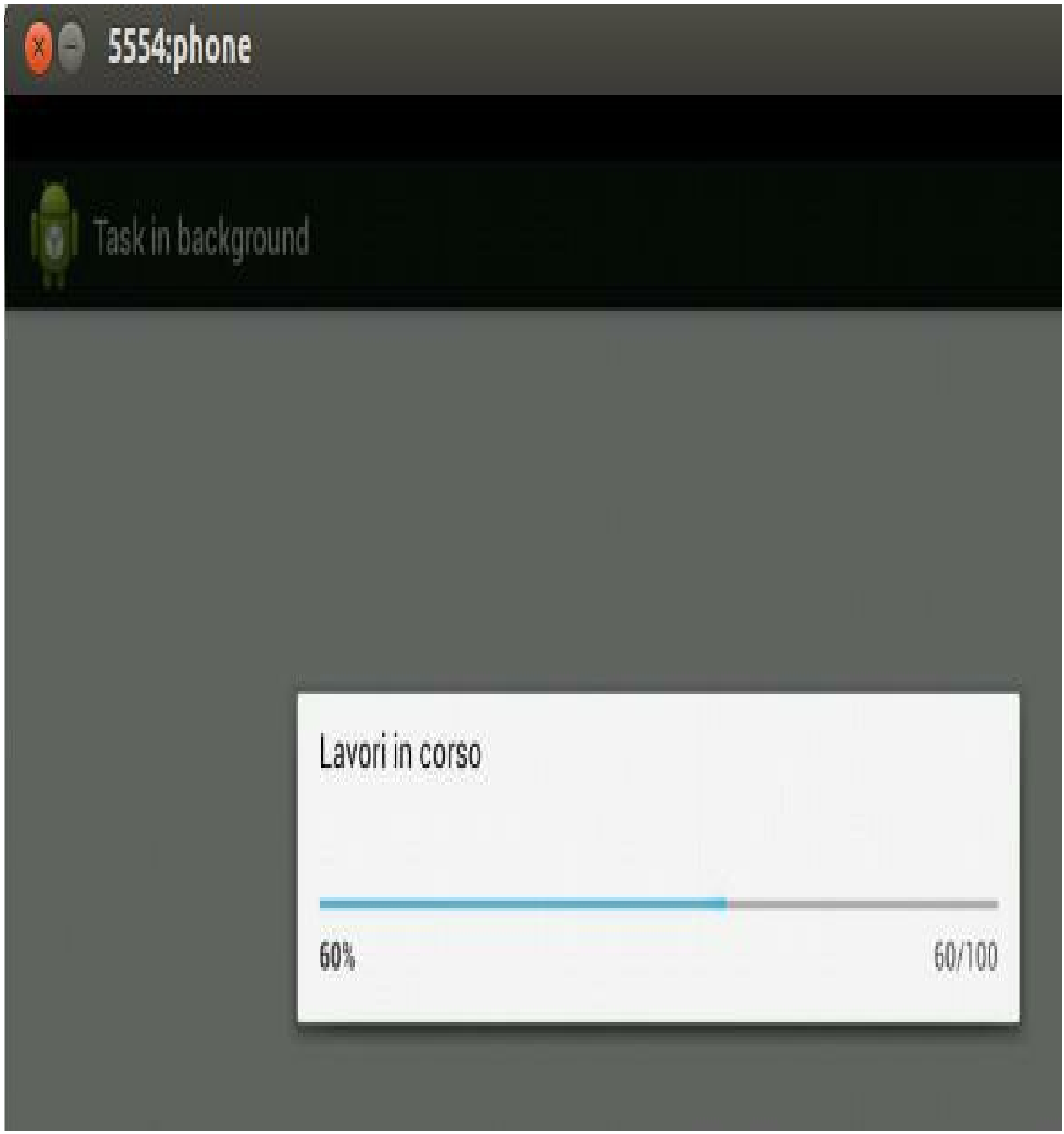
- `onPreExecute`: inizializza le operazioni prima che avvenga l'esecuzione di doInbackground. Nello specifico prepara la ProgressDialog e la mostra;
- `onPostExecute`: viene eseguito alla fine di doInBackground ed

anch'esso svolge operazioni collegate all'interfaccia utente, provocando l'apparizione di un Toast;

- `onProgressUpdate`: viene invocato ogni volta che dall'interno di doInBackground viene chiamato publichProgress. Serve a fornire aggiornamenti periodici all'interfaccia utente ed in questo caso a spostare la barra di progresso in avanti di un passo.

Un altro aspetto di AsyncTask cui si deve prestare attenzione sono i parametri della classe. Nell'esempio, **BackgroundTask estende la versione** `<Void, Integer, String>` **di AsyncTask**. Questi tre tipi di dato saranno, rispettivamente, il tipo di dato accettato in input dai metodi `doInBackground`, `onProgressUpdate`,`onPostExecute`.

L'immagine che segue mostra un momento di esecuzione dell'esempio



Il discorso su thread ed AsyncTask

non si è affatto esaurito, infatti saranno strumenti necessari già dai prossimi capitoli per la gestione di Service e attività in Rete.

Capitolo 33 – Lavoriamo in backgoud con i Service

Dal capitolo precedente si è fatta la conoscenza di AsyncTask che, a suo modo, permette di **avviare attività asincrone** a supporto dell'interfaccia utente. I task gestiti da questa classe non dovrebbero essere molto lunghi, "a few seconds at the most" ('pochi secondi al massimo') come dichiara esplicitamente la documentazione ufficiale.

Per **lavori di durata lunga o addiritura indeterminata**, si deve ricorrere ad un'appropriata componente Android, i **Service**.

Per utilizzare un Service è necessario

svolgere due operazioni:

- creare una classe Java che estenda Service o un suo derivato;
- registrare il service nel manifest con il nodo `<service>`

```
<service
android:name="LogService"/>
```

L'attributo android:name definisce quale classe Java implementa il service, in questo caso sarebbe la classe **LogService**.

Tipologie di Service

I Service sono classificabili in due tipologie, dipendentemente dal modo in cui vengono avviati:

- i service **Started** vengono avviati tramite il metodo `startService()`. La loro particolarità è di essere eseguiti in background indefinitamente anche se la componente che li ha avviati viene terminata. Generalmente non offrono interazione con il

chiamante e proseguono finchè non vengono interrotti con il metodo `stopService` o si auto-interrompono con `stopSelf();`

- i service **Bound** vivono in una modalità client-server. Hanno senso solo se qualche altra componente vi si collega. Vengono interrotti nel momento in cui non vi sono più client ad essi collegati.

Chiariamo subito che i service delle due categorie non sono radicalmente

diversi. Ciò che li distingue è il modo in cui vengono avviati ed i metodi di callback implementati al loro interno. Uno stesso service può essere avviato in maniera started o bound.

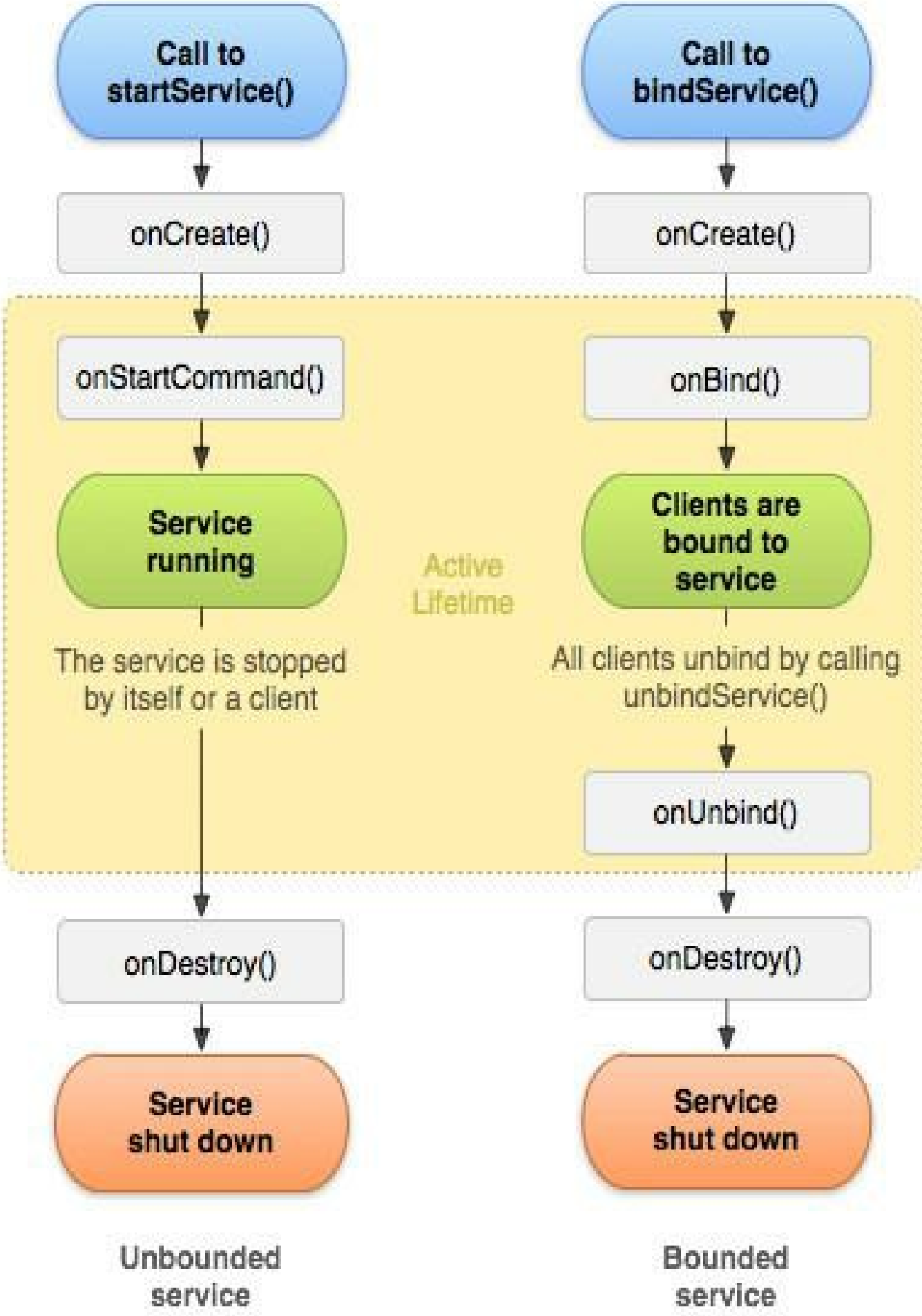
Emerge comunque una differenza nei tipi di lavori che sono più consoni all'uno o all'altra categoria.

I **Service Started** sono da predilligere per operazioni con una loro finalità indipendente dallo stato delle altre applicazioni. Si potrebbero occupare di aggiornamenti dati in background, scaricamento di file o immagini, sincronizzazione remota verso server esterni, etc. Considerando che il service Started rimarrà in background a lungo la sua esistenza deve essere

giustificata dalla finalità preposta.

I **Service Bound** svolgono il ruolo di supporto ad altre applicazioni. Non rischiano pertanto di essere "dimenticati" in background come potrebbe malauguratamente succedere agli Started ma non sono adatti a lavori da eseguire continuamente in background.

La differenza tra le due tipologie si riflette anche sul **ciclo di vita**. Il diagramma (fonte: documentazione ufficiale Android) seguente li mette a confronto:


Call to startService()
Call to bindService()
onCreate()
onCreate()
onStartCommand()
onBind()
Service running
Clients are bound to service
Active Lifetime
The service is stopped by itself or a client
All clients unbind by calling unbindService()
onUnbind()
onDestroy()
onDestroy()
Service shut down
Service shut down
Unbounded service
Bounded service

Nell'immagine, sfilano le fasi attraversate da un Service Started (sulla sinistra) e da uno Bound (sulla destra). Entrambi i cicli di vita iniziano e terminano con i metodi di callback `onCreate` e `onDestroy`. Le differenze si concentrano nella fase in cui il Service viene attivato. Mentre l'avvio di un service Started viene notificato per mezzo di `onStartCommand` l'inizio e la fine della connessione con un service bound viene segnalato dai metodi `onBind` e `onUnbind`.

Service e Thread

Nel ciclo di vita dei Service, non c'è alcun metodo che viene eseguito in background. Non si trova traccia di qualcosa che ricordi il *doInBackground* di AsyncTask o il *run* dei Thread. Questo perchè **il Service, di suo, non possiede alcun thread.** Fondamentalmente, il suo funzionamento è "sincrono".

Per permettere l'attività asincrona del service è necessario fornirlo almeno di un thread secondario. Quindi si ripresenta il problema paventato nel capitolo precedente: il thread deve essere fornito mediante estensione della classe Thread o avvio di Executors.

Queste sono operazioni che possono essere svolte bene ed in maniera efficiente con un po' di esperienza ma un neo-programmatore potrebbe risultarne scoraggiato. Android, come al solito, offre un'alternativa "pratica" anche in questo caso. Si può usare un discendente di Service, IntentService, che nasce già con un thread incorporato.

Al di fuori delle operazioni in background – da collocare all'interno del metodo onHandleIntent – l'IntentService non richiede molto lavoro. L'unico altro metodo obbligatorio da creare è un costruttore senza parametri.

LogService: il primo servizio

Un esempio classico che permette un rapido approccio ai Service è quello di creare un servizio di log avviato mediante Activity.

Prendiamo un'interfaccia utente dotata di due pulsanti, "Avvia" e "Arresta". Come il nome lascia presagire il primo avvia un Service, il secondo l'arresta.

Questo il layout:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/re

    android:layout_width="match_parent"

    android:layout_height="wrap_content">
```

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Avvia"
    android:onClick="startService"/>
  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Arresta"
    android:onClick="stopService"/>
</LinearLayout>
```

questo invece il codice dell'Activity:

```
public class MainActivity extends Activity
{
    @Override
        protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void startService(View v)
    {
                                    startService(new Intent(this,LogService.class));
    }
```

```
    public void stopService(View v)
    {
                                        stopService(new
Intent(this,LogService.class));
    }

}
```

Niente di nuovo in entrambi oltre all'uso dei metodi startService e stopService i quali, come spiegato, denunciano l'utilizzo di un Service avviato in modalità Started.

Molto importante: non dimenticare di dichiarare il Service

nell'AndroidManifest.

Il Service utilizzato è di tipo IntentService. Non dovremo fornirlo di altro se non di un costruttore senza parametri in input e dell'implementazione di un metodo `onHandleIntent`:

```
public class LogService extends IntentService
{
    public LogService()
    {
        super("LogService");
    }

    @Override
```

```
protected void onHandleIntent(Intent i)
{
    int n=0;
    while(true)
    {
            Log.i("PROVA SERVICE", "Evento
n."+n++);
        try {
            Thread.sleep(10000);
        }
        catch (InterruptedException e)
        { }
    }
}
```

```
    @Override
    public void onDestroy()
    {
        Log.i("PROVA SERVICE", "Distruzione Service");
    }
}
```

L'esecuzione in background del thread secondario produrrà l'immissione di messaggi di log in Logcat sfruttando il metodo *i* della classe Log. Il risultato saranno messaggi simili, nel formato, a quelli visibili in figura:

I	05-10 22:56:43.394	11810	11893	PROVA SERVICE	Evento n.5
I	05-10 22:56:53.394	11810	11893	PROVA SERVICE	Evento n.6
I	05-10 22:57:03.394	11810	11893	PROVA SERVICE	Evento n.7

Ultima nota, il Service è stato fornito di un'implementazione di onDestroy, chiamata in causa al momento dell'arresto del servizio, che aggiunge un messaggio finale di log.

Capitolo 34 – Androird e la rete

In un corso Android, che si tratti di tutorial scritti, videocorsi o lezioni frontali, non si può negare che uno dei momenti più attesi dai discenti sia la lezione che riguarda l'accesso alla Rete. Non è la prima volta che in questa guida si parla di accesso alla Rete. È già successo a proposito della WebView, il controllo che permette di integrare web app in un'interfaccia utente. Quello che viene presentato in questo capitolo è un discorso più ampio, un insieme di strumenti che permettono all'applicazione di consultare, scaricare ed inviare dati in Internet.

Prerequisiti

A proposito di connessioni remote, sono due i concetti da tenere a mente, entrambi già incontrati nel corso della guida e fondamentali per evitare errori ed eccezioni di sicurezza:

- nel file AndroidManifest.xml va indicata la **permission Internet** apponendo il seguente nodo: `<uses-permission android:name="androi`

- l'attività in Rete deve essere **eseguita in**

modalità asincrona, su un thread separato. In questo caso, senza scomodare i Service, ci si potrà avvantaggiare della classe AsyncTask, utilizzando tutto il codice che verrà presentato nel metodo doInBackground.

Leggere contenuti in Rete

Per eseguire l'accesso alla Rete, esistono fondamentalmente due modi in Android: usare classi Java standard o librerie esterne incluse nel sistema.

Il tipo di interazione remota più comune è quello via HTTP, il protocollo comunemente usato per la distribuzione dei contenuti Internet. Questo protocollo può essere trattato con la classe **HttpUrlConnection**, tipica del mondo Java, o con la libreria **HttpClient** gestita dalla fondazione Apache.

Per programmatori Java entrambi i casi sono generalmente piuttosto noti. Ma quale usare? Uno dei due è migliore dell'altro? Entrambe, al giorno d'oggi,

possono considerarsi funzionali. Sicuramente la libreria di Apache è molto estesa e offre tantissime funzionalità ma la classe HttpUrlConnection è più che sufficiente nella maggior parte dei casi come ad esempio nel normale scaricamento di un file. Quindi, di norma, si può usare la HttpUrlConnection tranne nei casi in cui servano funzionalità differenti disponibili solo in HttpClient.

Mettiamo a confronto gli utilizzi.

Immaginiamo che all'indirizzo *http://www.mioserver.it/storage/file.txt* sia disponibile un file di cui si deve fare il download.

Con HttpUrlConnection si farebbe:

URL url=new URL("http://www.mioserver.it/storage/file.txt");

recuperando direttamente lo stream ai contenuti:

InputStream is= url.openStream();

oppure passando per la connessione con:

HttpURLConnection conn= (HttpURLConnection) url.openConnection();

InputStream is= conn.getInputStream();

Con HttpClient, in alternativa, si tratta di preparare una *request* HTTP ed eseguirla ottenendo in cambio una *response*:

HttpClient request=new DefaultHttpClient();

HttpGet get=new HttpGet("http://www.mioserver.it/storage/file.txt");

HttpResponse response=request.execute(get);

InputStream is=response.getEntity().getContent()

In tutti i casi, si è ottenuto un riferimento ad un InputStream. Per suo tramite, si potrà fare accesso ai contenuti del file. Grazie all'astrazione offerta dagli Stream i contenuti potranno essere recuperati come se il file fosse locale. La classe che svolgerà il lavoro dipende dalla tipologia del formato dei dati: binari o testuali.

Se fosse un file testuale da scaricare in uno StringBuffer potremmo mettere in

pratica questo codice:

```
BufferedReader r=new BufferedReader(new InputStreamReader(is));

String s=null;

StringBuffer sb=new StringBuffer();

while((s=r.readLine())!=null)

{

    sb.append(s);

}
```

Trasmettere dati in Rete

Se si desidera inviare dati ad un server mediante HTTP, è probabile che lo si voglia fare con il metodo POST. Parafrasando il codice visto con HttpClient, un'operazione del genere si può portare a termine così:

```
HttpClient client = new DefaultHttpClient();

HttpPost post = new HttpPost(url);

List<BasicNameValuePair> parametri = <strong>new</strong>
ArrayList<BasicNameValuePair>(1);

parametri.add(new
BasicNameValuePair("cognome", "Rossi"));

parametri.add(new
BasicNameValuePair("nome", "Sergio"));

parametri.add(new BasicNameValuePair("eta",
```

```
"21"));
    post.setEntity(new
UrlEncodedFormEntity(parametri));
    HttpResponse resp = client.execute(post);
```

A differenza delle operazioni di lettura, si è usata la classe HttpPost in sostituzione di HttpGet. I dati da trasmettere vengono codificati in una lista di BasicNameValuePair. Ognuno di questi oggetti è costituito da una coppia chiave/valore. I parametri nel complesso alla fine diventano l'Entity della richiesta HTTP con l'invocazione di setEntity.

Appena sistemati i parametri, il funzionamento di HttpClient procede

come abbiamo già visto. La richiesta viene eseguita e della risposta ottenuta si può leggere il contenuto nell'Entity come Stream.

Un caso particolare: il DownloadManager

La lettura da remoto, come visto sinora, è ottimale per scambi di informazioni finalizzati a stabilire una comunicazione tra app e server. Generalmente, i dati ricevuti non hanno una mole eccessiva e sono codificati in un formato comune come JSON, XML, testo o CSV per agevolarne il parsing e l'utilizzo nelle app.

Per **download di dimensioni piuttosto grandi**, sarebbe opportuno predisporre un funzionamento più "robusto". Non è necessario reinventare la ruota in quanto esiste un servizio di sistema che già svolge bene questo lavoro: il

DownloadManager.

Come tutti i servizi di sistema, se ne deve **recuperare un riferimento**:

DownloadManager manager= (DownloadManager) getSystemService(DOWNLOAD_SERVICE);

A questo punto si **crea una richiesta** contenente l'indirizzo del file da scaricare e la si accoda nel manager:

Request request=new Request(Uri.parse("http://.... "));

manager.enqueue(request);

In funzione delle richieste pendenti, il servizio farà il download non appena possibile.

Questo servizio non solo offre "già

pronta" una funzionalità molto utile ma è **altamente configurabile**. Infatti si può scegliere, tra l'altro, se mostrare una barra di avanzamento nell'area delle notifiche, la posizione di salvataggio del file all'interno dello Storage e, cosa molto importante, se svolgere il download con ogni tipo di connessione o solo in presenza di Wi-Fi.

Le potenzialità e la flessibilità del DownloadManager lo rendono un servizio utilissimo tanto da poter considerare assolutamente inutile, se non addirittura dannoso, creare in proprio una soluzione alternativa per il medesimo scopo.

Capitolo 35 – Consumare servizi REST da Androird

Oltre al puro scaricamento di file, l'interazione con la Rete offre grandissime potenzialità. Una fra tutte: lo sfruttamento di servizi Web. Ormai non è più una novità parlare di Web Services. Si tratta essenzialmente di funzionalità rese disponibili in Rete da servizi remoti dai quali client distribuiti nel mondo possono recuperare informazioni, richiedere elaborazioni e molto altro ancora. Uno stile di servizio Web che si è diffuso molto già da diversi anni è REST (REpresentational State Transfer). Il motivo di ciò è la semplicità con cui può essere

implementato e la diffusione dei concetti che ne sono alla base.

Introduzione ai servizi REST

Il funzionamento di questi servizi si basa sulla possibilità di sfruttare risorse disponibili in Rete mediante i classici **metodi del protocollo HTTP**. Di questi, i più comuni sono GET e POST, vecchia conoscenza degli sviluppatori web: il primo concepito per leggere dati da remoto senza apportare modifiche, il secondo per inviare dati verso il servizio con lo scopo di richiederne l'inserimento nella base dati. Oltre a questi, HTTP possiede altri metodi ed in particolare due verranno coinvolti nel discorso: PUT per richiedere la modifica dei dati e DELETE per averne

la cancellazione.

Si noti che questi quattro metodi – POST, GET, PUT, DELETE – richiamano gli **stessi quattro concetti espressi dai metodi CRUD dei database**: creazione, lettura, modifica e cancellazione.

In questo senso i servizi REST possono essere visti come **un modo distribuito per gestire un database**.

Inoltre per ognuno dei suddetti metodi, il servizio Web specificherà con apposita documentazione quali URL devono essere contattati.

I ContentProvider, visti nei capitoli precedenti, da un punto di vista concettuale possono essere considerati

un'adozione della mentalità REST nella condivisione di informazioni nel sistema Android.

Altro concetto appartenente al protocollo HTTP, sono i **codici di stato** contenuti nella risposta. Tra i più comuni ricordiamo:

200	OK
400	Bad Request
403	Forbidden
404	Not found
500	Internal Server error

Avranno un ruolo importante nei servizi REST ma appartengono anche

alla comune esperienza della navigazione Internet: quante volte l'invocazione di un indirizzo web non corretto causava l'apparizione nel browser del classico messaggio "Error 404 – Page not found"?

In generale, al di là dei singoli codici, è importante ricordare che se la prima cifra del codice è 2 significa che comunque l'esecuzione è andata in porto, se è 4 indica errore da parte del client nella richiesta, se è 5 indica l'occorrenza di un errore dal lato server.

Le risorse offerte da servizi REST possono essere rappresentati in una moltitudine di **formati**, non ce n'è uno ufficiale. L'ideale sarebbe usare non

solo formati molto diffusi – XML o JSON – ma possibilmente prevedere la distribuzione della stessa risorsa in vari formati permettendo così la fruizione nella modalità preferita.

Nel prosieguo dell'articolo si approfondirà uno dei formati più comuni per servizi REST, ne vedremo le modalità di utilizzo in Android ed il suo impiego pratico in un servizio web. Stiamo parlando di JSON.

JSON in Android

JSON (Javascript Object Notation) è un formato stringa per la rappresentazione di dati organizzati in oggetti e array. Negli ultimi anni, soprattutto grazie alla sua semplicità e al suo largo impiego in Ajax, ha acquisito una notevolissima popolarità soprattutto a discapito di XML. In Android, se ne può fare uso e, come al solito, il sistema contiene tutto il necessario.

Molto spesso le stringhe in JSON contengono array di oggetti. Per farne il parsing si può apprezzare la comodità delle classi **JSONArray** e **JSONObject** disponibili nel package org.json.

Prendiamo la semplice stringa JSON che contiene tre oggetti. Ogni oggetto a sua volta contiene due stringhe, nome e cognome di una persona.

```
[
  {
    "nome":"Lucio",
    "cognome":"Bianchi"
  },
  {
    "nome":"Paolo",
    "cognome":"Neri"
  },
  {
```

```
    "nome":"Sergio",
    "cognome":"Rossi"
  }
]
```

Se nel nostro codice Android avessimo bisogno di acquisirne i dati, supponendo che l'array fosse nella stringa `json`, potremmo scrivere le seguenti righe:

```
JSONArray array=new JSONArray(json);
String persone=new String[array.length()];
for(int i=0;i<array.length();i++)
{
                                        String
```

```
nome=array.getJSONObject(i).getString("nome");
                                        String
cognome=array.getJSONObject(i).getString("cogno
    persone[i]=nome+" "+cognome;
}
```

Come si vede il costruttore di JSONArray implementa direttamente il parsing della stringa passata e l'oggetto ottenuto è consultabile quasi come un array vero e proprio mediante i metodi `length()`, per leggere la lunghezza, e getJSONObject che restituisce il JSONObject in una determinata posizione.

Ogni JSONObject ottenuto all'interno del ciclo può essere letto "in stile

mappa” recuperando i valori in base alla chiave assegnata in JSON. Con semplici modifiche, l’esempio precedente può essere adattato a molti tipi di parsing.

Interagire con un servizio REST

A questo punto dovremmo avere a disposizione tutti i prerequisiti necessari per l'interazione con un servizio REST.

Riepiloghiamoli:

- concetti basilari su **cos'è REST**: una visione chiara di come viene impiegato HTTP con i suoi metodi e codici di stato;
- saper gestire anche in maniera basilare il

parsing e la formattazione **JSON** o di altro formato che sia necessario usare;

- saper fare in modo che la nostra app riesca ad **accedere alla Rete e dialogare in HTTP**. Abbiamo visto almeno due modi per farlo: classe Java tradizionale HttpURLConnection, meno funzionalità ma molto comuni e libreria Apache HttpClient, completissima e già inclusa in Android;

- ricordare sempre le **due regole** base per l'accesso alla Rete da Android: dichiarare le **permission** adeguate (almeno android.permission.INTE ed eseguire gli **accessi da un thread secondario**. Per il secondo punto, nella maggior parte dei casi, è sufficiente saper usare AsyncTask.

titolo di esempio, immaginiamo che ci sia un servizio REST che all'URL *http://www.mioservizio.it/persone*, via

GET, restituisca in JSON un array di oggetti. Ogni oggetto rappresenta una persona specificandone nome,cognome ed età.

Vediamo le porzioni di Java che servirebbero a sfruttare entrambe le funzionalità. Ricordiamo per l'ennesima volta che il seguente codice va usato, previo inserimento della permission INTERNET nel manifest, all'interno di un metodo doInBackground di AsyncTask.

String url="http://www.mioservizio.it/persone";

String[] persone=null; // conterrà i risultati

HttpClient request=new DefaultHttpClient();

HttpGet get=new HttpGet(url);

```
HttpResponse response=request.execute(get);
responseCode=response.getStatusLine().getStatu
if (responseCode==200)
{
                                        InputStream
istream=response.getEntity().getContent();
    BufferedReader r=new BufferedReader(new
InputStreamReader(istream));
    String s=null;
    StringBuffer sb=new StringBuffer();
    while((s=r.readLine())!=null)
    {
      sb.append(s);
    }
```

```
                    JSONArray array=new
JSONArray(sb.toString());
        persone=new String[array.length()];
        for(int i=0;i&lt;array.length();i++)
        {
                                                String
nome=array.getJSONObject(i).getString(&quot;nor
                                                String
cognome=array.getJSONObject(i).getString(&quot
                                                String
eta=array.getJSONObject(i).getString(&quot;eta&
                                persone[i]=nome+&quot;
&quot;+cognome+&quot; di anni &quot;+eta;
        }
        return persone;
    }
```

Notare che, una volta effettuata la connessione ed ottenuta la risposta, è necessario leggere il codice di risposta HTTP. In questo caso, le operazioni di parsing vengono eseguite solo se il codice HTTP restituito è 200. Come spiegato in precedenza si possono attuare comportamenti differenti per codici di stato diversi.

Le operazioni di interpretazione dei risultati vengono effettuate aggregando gli elementi appresi in questa lezione e nelle precedenti:

- con `response.getEntity()`

viene prelevato il contenuto dell'entity HTTP sotto forma di InputStream;

- l'InputStream verrà letto iterativamente al fine di ritrovare tutto il suo contenuto all'interno di uno StringBuffer;
- o StringBuffer, quando completo, conterrà un array JSON formato da un elenco di persone e pertanto se ne potrà fare il parsing con JSONArray;
- dopo il parsing, i dati

prelevati dal servizio, saranno disponibili in una struttura dati Java – in questo caso un semplice array di stringhe – e quindi pronti per essere sfruttati nella propria app.

Gli altri metodi HTTP

L'esempio ha mostrato solo una lettura di dati via GET. Gli altri casi non vengono esemplificati nei dettagli ma si consideri che la libreria HttpClient oltre a `HttpGet` contiene classi corrispondenti agli altri metodi HTTP: `HttpPost`, `HttpPut` e `HttpDelete`.

Inoltre, mentre eventuali parametri nel GET vengono inviati concatenati all'URL in una *querystring*, negli altri metodi lo si può fare con liste di BasicNameValuePair come mostrato nel capitolo relativo all'accesso in Rete.

Lo studio può comunque essere proseguito sia sulla documentazione Android che della fondazione Apache in

entrambi i casi molto completa.

Capitolo 36 – Sensori e SensorManager in Android

Molto spesso giocando, usando applicazioni o semplicemente tenendo in mano un device ci è capitato di notare che il dispositivo “si accorge” di una serie di fattori ed eventi fisici: se lo giriamo, se lo scuotiamo e via dicendo.

Altre volte si può essere rimasti stupiti notando che alcune app forniscono informazioni sull’ambiente in cui ci troviamo. Percepiscono magari temperatura, umidità, luminosità.

“Ma come fa a saperlo?”: la domanda nasce spontanea.

Questo capitolo vuole mostrare che

queste funzionalità non celano magie ma un po' di elettronica amalgamata con software ben fatto.

I dispositivi Android grazie ai sensori di cui sono forniti riescono a percepire movimenti, condizioni ambientali e lo faranno sempre più e sempre con maggiore precisione grazie all'ampliamento costante di queste tecnologie. A noi spetta la parte, se vogliamo, più divertente di tutto questo: leggere facilmente queste informazioni con le API del framework e usarle per arricchire le nostre app.

Classificazione dei sensori

Iniziamo con un po' di classificazioni. Innanzitutto, i sensori possono essere suddivisi in tre grandi gruppi:

- sensori di **movimento**: percepiscono le forze fisiche che agiscono sul dispositivo. Ad esempio, l'accelerometro, il giroscopio, sensore di gravità;
- sensori **ambientali**: rilevano particolari dell'ambiente in cui ci si trova: temperatura,

pressione, umidità;

- sensori di **posizione**: raccolgono dati sulla posizione del dispositivo, ad esempio il sensore di orientamento.

Inoltre, i sensori, dipendentemente dal modo in cui sono implementati, possono essere **hardware** o **software**. I primi corrispondono a dei veri e propri elementi elettronici inseriti nel dispositivo. I secondi sono delle elaborazioni basate sui dati raccolti dai sensori hardware. Quelli software sono chiamati anche virtuali in quanto possono essere consultati con lo stesso

interfacciamento di quelli hardware dissimulando quindi la loro natura software.

Alcuni sensori devono essere necessariamente hardware, altri esistono solo software mentre alcuni possono essere hardware o software a seconda dell'implementazione che è stata scelta per il particolare dispositivo.

Il SensorManager

Come in molte altre situazioni, il sottosistema Android che vogliamo sfruttare ci viene dischiuso da un *system service*, accessibile mediante una classe "manager".

In questo caso, si tratta del SensorManager.

Per ottenerne un riferimento, procediamo ad un passo ormai di rito:

```
private SensorManager mSensorManager;

. . .

. . .

mSensorManager =
(SensorManager)getSystemService(SENSOR_SERV
```

La prima cosa che può essere utile fare con il SensorManager è chiedergli un inventario dei sensori disponibili nel nostro dispositivo.

Usando una serie di costanti intere (tutte ben spiegate nella documentazione ufficiale) si può chiedere una lista dei sensori:

```
<Sensor>
sensors=mSensorManager.getSensorList(Sensor.TY
```

oppure verificare se un sensore è disponibile:

```
Sensor
ss=mSensorManager.getDefaultSensor(Sensor.TYP
```

Un po' tutti i dispositivi avranno a

disposizione almeno tre o quattro sensori essenziali per la vita di uno smartphone tra cui accelerometro, orientamento e rotazione.

Leggere dati da un sensore

La prassi comune per ricevere dati periodici da un sensore è **registrare un listener** nella nostra applicazione. Ciò, da un punto di vista sintattico, obbligherà all'implementazione di un metodo di callback all'interno del quale si potrà fare un qualche uso delle misurazioni rilevate.

Un tipico schema di Activity che legge dati da un sensore potrebbe essere questo:

```
public class MainActivity extends Activity implements SensorEventListener
{
    private SensorManager mSensorManager;
```

```
    private Sensor sensor;



    protected void onResume() {
        super.onResume();
    mSensorManager.registerListener(this,  sensor,
SensorManager.SENSOR_DELAY_NORMAL);
    }

    protected void onPause() {
        super.onPause();
        mSensorManager.unregisterListener(this);
    }

    @Override
```

```
protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
                                mSensorManager =
(SensorManager)getSystemService(SENSOR_SERV
                                sensor =
mSensorManager.getDefaultSensor(
        /*
                * Costante relativa al sensore da
monitorare
        * */
                );
    }
```

```
    @Override
      public void onSensorChanged(SensorEvent
event)
    {
       /*
         * Codice di gestione dei nuovi eventi del
sensore
       * */
    }

    @Override
      public void onAccuracyChanged(Sensor s,
int i)
    {
```

```
    }
}
```

Gli aspetti da notare maggiormente sono:

- nel metodo `onCreate` è stato prelevato un riferimento al SensorManager. Opzionalmente questo punto sarà buono per recuperare un riferimento anche al sensore specifico con cui si vuole interagire;

- nei metodi `onPause` e `onResume` che come sappiamo regolano l'inizio e la fine dell'interazione tra Activity e utente avviene, rispettivamente, la registrazione e la cancellazione del listener;
- l'activity implementa l'interfaccia SensorEventListener che forza all'override di due metodi `onAccuracyChanged` e `onSensorChanged`.

Il metodo **onSensorChanged** costituisce il cuore dell'interazione con il sensore. È qui che arrivano le chiamate del listener ogni volta che sono disponibili nuove misurazioni. L'evento notificato verrà formalizzato con un oggetto di classe `SensorEvent`.

SensorEvent permette di leggere i valori recuparati come un array numerico. Il tutto visto in questo modo potrebbe sembrare semplice. La difficoltà sta proprio nell'interpretare e sfruttare i valori dell'evento. Essendo i sensori dei misuratori di grandezza fisiche, i dati letti con essi dovrebbero essere sottoposti ad opportune valutazioni nel rispetto, eventualmente, di leggi scientifiche. In un capitolo

successivo, si avrà modo di sperimentare l'accelerometro ed in quel caso dovremo interpretare delle accelerazioni.

Capitolo 37 – Geolocalizzazione con Android

L'utente passa giornate spostandosi e raccogliendo informazioni – più o meno volutamente – nel proprio device Android. Consultazioni internet, appunti, eventi calendario, foto, chiamate, messaggi. Il dispositivo diventa una specie di "diario errante" dell'esperienza di vita quotidiana. La possibilità di associare informazioni geografiche a questi ricordi apre scenari nuovi e ciò basta a giustificare la rapidissima diffusione che ha avuto nell'informatica mobile la geolocalizzazione.

Intendiamo con questo termine la capacità di un dispositivo di rilevare la propria posizione geografica nel mondo reale. Non stiamo parlando ormai di una dote rara, quasi ogni smartphone o tablet oggi contiene dei sistemi di localizzazione.

I più comuni sono:

- **network-based**: rileva le reti mobili Wi-Fi e GSM disponibili nella zona ed in base a questi calcola la propria posizione. Non molto accurato ma immancabile nei dispositivi;

- **GPS** (Global Positioning System): acronimo famosissimo, si basa sull'intercettazione di messaggi inviati da satelliti che ruotano attorno alla Terra. Tali comunicazioni contengono l'informazione oraria ed altri dati relativi all'orbita percorsa. Il dispositivo intercettando i segnali di almeno quattro di questi satelliti con l'applicazione di formule matematiche riesce a calcolare la

propria posizione. Accurato e diffusissimo tranne che in alcuni dispositivi di fascia bassa. Praticamente il sistema di localizzazione per antonomasia.

Esempio pratico: GPS nell'Activity

Entriamo subito nel vivo creando un'Activity che richiede informazioni GPS e le mostra nel suo layout. Oltre a latitudine e longitudine l'Activity mediante un oggetto denominato GeoCoder recupererà l'indirizzo cui corrisponde la posizione.

Il layout dell'activity è una griglia molto semplice. TableLayout con una serie di campi di testo da completare:

```
<TableLayout
xmlns:android="http://schemas.android.com/apk/re

    android:layout_width="wrap_content"

    android:layout_height="wrap_content">
```

```
        <TableRow android:padding="5dp">
                <TextView  android:text="Abilitato"
android:padding="5dp"
                android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
              <TextView  android:id="@+id/enabled"
android:padding="5dp"
                android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
        </TableRow>
        <TableRow android:padding="5dp">
                <TextView  android:text="Data  ora"
android:padding="5dp"
                android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
              <TextView android:id="@+id/timestamp"
android:padding="5dp"
```

```
                android:layout_width="wrap_content"
android:layout_height="wrap_content"/>

        </TableRow>

        <TableRow android:padding="5dp">

                <TextView  android:text="Latitudine"
android:padding="5dp"

                android:layout_width="wrap_content"
android:layout_height="wrap_content"/>

                <TextView  android:id="@+id/latitude"
android:padding="5dp"

                android:layout_width="wrap_content"
android:layout_height="wrap_content"/>

        </TableRow>

        <TableRow android:padding="5dp">

                <TextView  android:text="Longitudine"
android:padding="5dp"
```

```
                android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
            <TextView android:id="@+id/longitude"
android:padding="5dp"
                android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
        </TableRow>
        <TableRow android:padding="5dp">
                <TextView  android:text="Località"
android:padding="5dp"
                android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
                <TextView  android:id="@+id/where"
android:padding="5dp"
                android:lines="2"
                android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
```

```
    </TableRow>

</TableLayout>
```

Da ricordare che per l'accesso ai dati GPS è necessaria un'apposita **permission.** Nel manifest andremo ad inserire questa riga:

```
<uses-permission
android:name="android.permission.ACCESS_FINE
```

Questa permission va bene sia per usare il GPS sia per la localizzazione network-based. Qualora si volesse usare solo quest'ultima è sufficiente la permission `ACCESS_COARSE_LOCATION`.

All'interno dell'Activity dovremo per

prima cosa registrare un Listener presso il LocationManager e lo faremo nel metodo `onResume`. Tale istanza sarà annullata in `onPause`.

```
public class MainActivity extends Activity
{
        private String providerId = LocationManager.GPS_PROVIDER;
    private Geocoder geo = null;
        private LocationManager locationManager=null;
    private static final int MIN_DIST=20;
        private static final int MIN_PERIOD=30000;

    private LocationListener locationListener =
```

```
new LocationListener()
        {
            . . .
            . . .

        };

        @Override
                protected void onCreate(Bundle
savedInstanceState)
        {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
        }
```

```
@Override
protected void onResume()
{
    super.onResume();
                    geo=new  Geocoder(this,
Locale.getDefault());
        locationManager = (LocationManager)
getSystemService(LOCATION_SERVICE);
                    Location  location  =
locationManager.getLastKnownLocation(LocationM
    if (location!=null)
        updateGUI(location);
                if  (locationManager!=null  &&
locationManager.isProviderEnabled(providerId))
        updateText(R.id.enabled, "TRUE");
```

```
        else
            updateText(R.id.enabled, "FALSE");
        locationManager.requestLocationUpdates(
MIN_PERIOD,MIN_DIST, locationListener);
    }

    @Override
    protected void onPause()
    {
        super.onPause();
                if (locationManager!=null &&
locationManager.isProviderEnabled(providerId))
            locationManager.removeUpdates(locatio
    }
```

. . .

. . .

}

Notare che, nell'onResume, il metodo **requestLocationUpdates** effettua la vera registrazione del listener. I parametri che utilizza sono:

- l'id del provider: la costante stringa che individua il tipo di provider da usare;
- il minimo intervallo di tempo, in millisecondi, che deve trascorrere tra

aggiornamenti della posizione;

- la minima distanza in metri che deve intercorrere tra due misurazioni;
- l'oggetto che svolge il ruolo di listener. Lo vedremo subito.

L'oggetto listener registrato viene implementato come classe interna all'Activity:

```
private LocationListener locationListener = new
LocationListener()
    {
        @Override
```

```
        public void onStatusChanged(String
provider, int status, Bundle extras)
        {


        }
        @Override
            public void onProviderEnabled(String
provider)
        {
            // attivo GPS su dispositivo
            updateText(R.id.enabled, "TRUE");
        }
        @Override
            public void onProviderDisabled(String
provider)
```

```
        {
            // disattivo GPS su dispositivo
            updateText(R.id.enabled, "FALSE");
        }
        @Override
         public void onLocationChanged(Location
location)
        {
            updateGUI(location);
         }
    };
```

I primi tre metodi – `onStatusChanged`, `onProviderEnabled`,

`onProviderDisabled` – notificano, rispettivamente, se il provider è disponibile o meno, se è abilitato, se è stato disabilitato.

L'ultimo metodo `onLocationChanged` **è il cuore del listener** e viene invocato ogni volta che nuove informazioni di posizione sono state recapitate.

L'oggetto Location contiene tutto ciò che è stato appreso dall'ultima misurazione del posizionamento e viene inviata al metodo `updateGUI` per riflettere gli aggiornamenti sulla interfaccia utente:

```
private void updateGUI(Location location)
    {
```

```
                Date timestamp = new
Date(location.getTime());
                updateText(R.id.timestamp,
timestamp.toString());
        double latitude = location.getLatitude();
                updateText(R.id.latitude,
String.valueOf(latitude));
                double longitude =
location.getLongitude();
                updateText(R.id.longitude,
String.valueOf(longitude));
        new AddressSolver().execute(location);
    }

    private void updateText(int id, String text)
    {
```

```
                TextView textView = (TextView)
findViewById(id);

        textView.setText(text);

    }
```

All'interno di `updateGUI`, oltre al codice di modifica delle TextView, è presente l'invocazione al **Geocoder** per la conversione delle coordinate in un indirizzo vero e proprio. Il Geocoder viene consultato in maniera asincrona mediante `AsyncTask`. Nel metodo doInBackground, la Location sarà convertita in una stringa frutto della concatenazione delle informazioni reperite:

```
private class AddressSolver extends
```

```
AsyncTask<Location, Void, String>
        {


            @Override
                                        protected      String
doInBackground(Location... params)
            {
                Location pos=params[0];
                double latitude = pos.getLatitude();
                double longitude = pos.getLongitude();


                List<Address> addresses = null;
                try
                {
                                                addresses    =
```

```
geo.getFromLocation(latitude, longitude, 1);
            }
            catch (IOException e)
            {


            }
            if (addresses!=null)
            {
                if (addresses.isEmpty())
                  {
                    return null;
                  }
                  else {
                     if (addresses.size() > 0)
```

```
                {
                        StringBuffer address=new
StringBuffer();
                    Address tmp=addresses.get(0);
                                for (int
y=0;y<tmp.getMaxAddressLineIndex();y++)
                        address.append(tmp.getAddres
                    return address.toString();
                }
            }
        }
        return null;
    }

    @Override
```

```
        protected void onPostExecute(String result)
    {
        if (result!=null)
            updateText(R.id.where, result);
        else
            updateText(R.id.where, "N.A.");

    }
}
```

Capitolo 38 – Accelerometro: come utilizzarlo

Uno dei sensori più ampiamente diffusi nei dispositivi Android è di sicuro l'**accelerometro**. Quindi, con riguardo alla sua ampia diffusione ed importanza, in questo capitolo sarà oggetto di un esempio che per quanto semplice potrà apparire dotato di una certa valenza pratica.

Esempio: usare lo “shake”

In questo esempio vedremo come intercettare un **evento di shake** e collegarvi una reazione. Per shake intendiamo genericamente l’atto di scuotere il dispositivo indipendentemente dalla direzione. Le accelerazioni fisiche impresse sull’hardware vengono captate dall’accelerometro. Il nostro compito sarà quello di capire se la loro intensità complessiva è tale da potervi riconoscere l’avvenimento di uno shake. Solo in questo caso attiveremo la reazione.

Nell’esempio, il layout è costituito da un solo form. Se durante la compilazione

viene effettuato uno shake, il dispositivo chiederà a mezzo finestra di dialogo se si vuole procedere alla cancellazione dei valori inseriti nei campi.


Accelerometer
Username:
topolino
Password:
Salva
Annulla
Shake
Username:
topolino
Password:
Pulire il form?
No
Si

Il codice del layout, come presumibile, non offre grandi novità. Questo il contenuto del file *res/layout/activity_main.xml*:

```
<TableLayout
xmlns:android="http://schemas.android.com/apk/re

    android:layout_width="match_parent"

    android:layout_height="match_parent">

   <TableRow >

      <TextView android:layout_weight="1"

            android:layout_height="wrap_conte
android:text="Username:"

            android:id="@+id/label1"/>

      <EditText android:layout_weight="5"

            android:inputType="text"
```

```
            android:layout_height="wrap_conte
            android:id="@+id/text1"/>
    </TableRow>
    <TableRow >
        <TextView android:layout_weight="1"
            android:layout_height="wrap_conte
android:text="Password:"
            android:id="@+id/label2"/>
        <EditText android:layout_weight="5"
            android:inputType="textPassword"
            android:layout_height="wrap_conte
android:id="@+id/text2"/>
    </TableRow>
    <TableRow >
        <Button android:id="@+id/button1"
```

```
                android:layout_width="wrap_content
                 android:layout_height="wrap_conte
                 android:text="Salva"/>
                                                <Button
android:layout_width="wrap_content"
                 android:layout_height="wrap_conte
                 android:text="Annulla"
                 android:id="@+id/button2"/>
    </TableRow>
</TableLayout>
```

“Ascoltare” il sensore

L’Activity dovrà svolgere il ruolo di listener per eventi del sensore:

```
public class MainActivity extends Activity implements SensorEventListener
{
    private SensorManager mSensorManager;
    private Sensor mAccelerometer;
    private float lastAcc = 0.0f;
    private float acceleration = 0.0f;
    private float totAcc = 0.0f;
    private boolean onEvent = false;

    . . .
    . . .
```

} // fine MainActivity

I membri privati contengono, in primis, riferimenti al *SensorManager* e ad un *Sensor* che in questo caso è l'accelerometro. Le altre variabili di tipo *float* serviranno per custodire valori temporanei nel calcolo delle accelerazioni.

Il metodo **onCreate** svolge le inizializzazioni assegnando il valore opportuno ai membri privati. Notiamo che le variabili *float* delle accelerazioni verranno impostate a valori corrispondenti alla gravità terrestre.

@Override

```
    protected void onCreate(Bundle
savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
                                mSensorManager =
(SensorManager)getSystemService(SENSOR_SERV
                                mAccelerometer =
mSensorManager.getDefaultSensor(Sensor.TYPE_A
        lastAcc=SensorManager.GRAVITY_EARTH;
        acceleration=SensorManager.GRAVITY_EAR
    }
```

Nel capitolo riguardante il ciclo di vita delle Activity si è spiegato a cosa servono i metodi *onResume* e *onPause*:

segnano, rispettivamente, l'inizio e la fine dell'interazione tra interfaccia e utente. Visto che la nostra Activity dovrà registrarsi per ricevere prontamente segnalazioni sugli eventi dell'accelerometro, per evitare di impiegare inutilmente risorse rinnoverà tale registrazione all'interno dell'*onResume* e la disdirà ad ogni *onPause*. Al di fuori dell'intervallo di tempo segnato da questi metodi, sarebbe assolutamente inutile oltre che vanamente dispendioso richiedere segnalazioni in merito.

```
protected void onResume() {

    super.onResume();

        mSensorManager.registerListener(this,
mAccelerometer,
```

```
SensorManager.SENSOR_DELAY_NORMAL);
    }

    protected void onPause() {
        super.onPause();
        mSensorManager.unregisterListener(this);
    }
```

L'implementazione dell'interfaccia *SensorEventListener* richiede l'override di due metodi:

- **onAccuracyChanged**: chiamato quando l'accuratezza del sensore

viene modificata. Nell'esempio non riceverà un'implementazione;

- **onSensorChanged**: come già appreso alcuni capitoli fa, riceve notifiche sugli eventi del sensore incarnati da un oggetto SensorEvent. Qui verrà inserito tutto il codice di gestione dell'evento e proprio dal SensorEvent leggeremo le accelerazioni ricevute.

@Override

```
    public void onAccuracyChanged(Sensor arg0,
int arg1)
    {  }

    @Override
    public void onSensorChanged(SensorEvent
event)
    {
        if (!onEvent)
        {
            float x = event.values[0];
            float y = event.values[1];
            float z = event.values[2];

            lastAcc = acceleration;
```

```
acceleration = x*x+y*y+z*z;
float diff = acceleration - lastAcc;
totAcc = diff*acceleration;
if (totAcc>15000)
{
    onEvent=true;
            AlertDialog.Builder  builder=new
Builder(this);
    builder.setMessage("Pulire il form?");
             builder.setPositiveButton("Sì",  new
OnClickListener()
    {
        @Override
         public void onClick(DialogInterface
arg0, int arg1)
```

```
                {
                    clean();
                    onEvent = false;

                }
            });
                builder.setNegativeButton("No",new
OnClickListener()
            {
                @Override
                public void onClick(DialogInterface
arg0, int arg1)
                {
                    onEvent=false;
```

```
                }

            });

            builder.show();

        }

    }

}

private void clean()

{

                    TextView    txt1=(TextView)
findViewById(R.id.text1);

                    TextView    txt2=(TextView)
findViewById(R.id.text2);

    txt1.setText("");
```

```
    txt2.setText("");
}
```

L'ulteriore metodo visibile nel codice, **clean**, verrà invocato quando si riterrà opportuno cancellare il form.

Tutto il riconoscimento dello shake si trova in queste righe

```
float x = event.values[0];
float y = event.values[1];
float z = event.values[2];


lastAcc = acceleration;
acceleration = x*x+y*y+z*z;
float diff = acceleration - lastAcc;
```

```
totAcc = diff*acceleration;

if (totAcc>15000)

{
```

Da *event* verranno lette le tre accelerazioni (una per ogni dimensione dello spazio). La risultante verrà calcolata sommandone i quadrati. La variabile *lastAcc* serve a salvare l'ultimo valore calcolato – quello che è avvenuto con l'evento precedente – mentre *acceleration* conterrà il nuovo valore. Infine in *totAcc* si cercherà di **valutare l'entità della variazione**. Se tale valore supera la soglia di 15000 assumeremo che lo shake sia avvenuto.

Il codice contenuto nel blocco if

quindi sarà la vera reazione allo shake. **L'Activity dovrà sapere sempre che il trattamento dell'evento è in corso**. Glielo dirà il valore di *onEvent* che verrà impostato a true non appena la variazione delle accelerazioni raggiungerà 15000.

Gli eventi dell'accelerometro verranno invocati molto spesso quindi dovremo stare attenti ad eseguire meno codice possibile e solo nei casi in cui sia strettamente necessario. Se si nota, non appena viene invocato *onSensorChanged* si verifica se è in corso la gestione di uno shake con:

if (!onEvent)

In caso positivo nulla sarà fatto per il momento. Ovviamente è fondamentale resettare *onEvent* impostandolo di nuovo a false nel momento in cui la finestra di dialogo viene chiusa.

Capitolo 39 – Touchscreen ed eventi Touch

Pensare un dispositivo Android senza touchscreen ormai è impossibile. La facoltà di un display di riconoscere il movimento di uno o più dita dell'utente su di sé è di fondamentale importanza. Ciò ha rappresentato una svolta generazionale nei device facilitando il pensionamento delle tastiere hardware e permettendo l'avvento di quelle software che noi tutti conosciamo.

In questo capitolo si inizierà a conoscere l'utilizzo del touch in un'app Android e le informazioni fondamentali che tale evento fornisce. Infine, con un esempio, se ne mostrerà un possibile

uso.

Abbiamo imparato a definire un listener ogni volta che la nostra app è in attesa di qualcosa. La gestione del touch non fa differenza. **Un metodo onTouchEvent riceverà le informazioni relative al singolo evento in un oggetto MotionEvent e predisporrà una reazione**.

A dimostrazione della fondamentale importanza che la gestione del touchscreen riveste non sarà necessario implementare alcuna interfaccia – come ad esempio si faceva con *SensorEventListener* per i sensori – ma il metodo *onTouchEvent* è "di serie" su qualsiasi View e sulle Activity.

Un oggetto *MotionEvent* contiene molte informazioni, ma quelle più "eloquenti" per il programmatore sono:

- l e **coordinate** del display in cui è avvenuto l'evento touch;
- l a **tipologia dell'evento** (i più comuni: *ACTION_DOWN*, generato quando il dito si è posato sullo schermo, *ACTION_MOVE*, se il dito si è mosso senza staccarsi dallo schermo, ed *ACTION_UP* nel caso

in cui il dito ha lasciato la superficie del display).

L’esempio

In questa dimostrazione riconosceremo mediante touch un click sullo schermo del dispositivo sfruttando il metodo onTouchEvent dell’Activity. Per fare ciò cercheremo una certa sequenzialità tra un evento ACTION_DOWN ed uno ACTION_UP. Con l’occasione, rilevando i tempi delle due fasi, si calcolerà approssimativamente quanti secondi è durato il click.

Il risultato finale è visibile nella figura in cui si vede un Toast che fornisce le informazioni raccolte con il piccolo esperimento.

Layout semplicissimo (file: res/layout/activity_main.xml):

<RelativeLayout

xmlns:android="http://schemas.android.com/apk/re

android:layout_width="match_parent"

android:layout_height="match_parent">

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:textStyle="bold"
    android:textSize="20dp"
     android:text="Tieni premuto il dito sul\n
display alcuni secondi"/>

</RelativeLayout>
```

Il codice dell'activity concentra le sue peculiarità in onTouchEvent:

```
public class MainActivity extends Activity
{
```

```
    private long inizio=0;
    @Override
            protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
      public boolean onTouchEvent(MotionEvent event)
    {
        float eventX = event.getX();
```

```
float eventY = event.getY();


switch (event.getAction())
{
    case MotionEvent.ACTION_DOWN:
        inizio=System.currentTimeMillis();
        break;
    case MotionEvent.ACTION_UP:
                                String posizione="
("+Math.round(eventX)+","+Math.round(eventY)+
                                long diffInSec=
(System.currentTimeMillis()-inizio)/1000;
                Toast.makeText(this, "Click durato
"+diffInSec+" secondi in posizione "+posizione,
Toast.LENGTH_SHORT).show();
        return true;
```

```
        }
        return false;
    }

}
```

I metodi di MotionEvent utilizzati sono:

- *getX()* e *getY()*: forniscono, rispettivamente, ascissa e ordinata della posizione del display in cui si è verificato l'evento;

- *getAction()*: restituisce una costante intera che permette di riconoscere il tipo di evento.

In caso di *ACTION_DOWN* viene salvato in una variabile intera dell'Activity l'informazione temporale in millisecondi. In *ACTION_UP* si farà la differenza tra l'attuale informazione oraria e quella salvata al momento di toccare lo schermo.

Capitolo 40 – Multitouch

Quanto visto nel capitolo precedente riguardava esclusivamente eventi scatenati da un singolo puntatore; o meglio, considerato che ci stiamo riferendo alla mano dell'utente, da un solo dito. Sono i cosiddetti eventi **single touch**. Molto frequenti sono comunque gli **eventi multitouch**, cioè quelli provocati dall'uso congiunto di più dita. Si pensi a quando, per rimpicciolire un'immagine, puntiamo due dita sul display e senza sollevarle le avviciniamo tra loro.

Concettualmente il multitouch nelle app Android presenta gli stessi fondamenti del suo "cugino" single.

Ugualmente si farà uso di:

- **TouchEventListener** per designare il componente incaricato di ricevere gli eventi relativi al touch;
- **onTouchEvent** sarà il metodo nel quale verranno recapitate le informazioni relative all'evento rilevato;
- **MotionEvent** sarà la tipologia di oggetto che veicolerà le informazioni acquisite come posizioni dei puntatori e tipologie

di evento.

La differenza starà nel fatto che dovranno essere rese contemporaneamente **disponibili tutte le informazioni sulla posizione dei singoli puntatori**. Ciò permetterà in questa sede di approfondire la struttura di *MotionEvent* che, probabilmente, quando si tratta il semplice single touch appare molto più elementare di quello che realmente sia.

Si consideri innanzitutto che tutte le dita poggiate contemporaneamente sul display costituiscono con i loro movimenti una "gestualità" complessiva. Il **MotionEvent assegnerà un ID ad ogni puntatore** e se ne servirà per

distinguere le informazioni raccolte per ognuno di essi.

All'interno del metodo *onTouchEvent* risulteranno molto utili i seguenti **metodi di MotionEvent**:

- **getPointerCount**: restituisce il numero di puntatori attualmente sul display;
- **getActionIndex**: fornisce l'indice del puntatore che viene trattato in questa singola invocazione di **onTouchEvent**;

- **getPointerId**: restituisce l'id del singolo puntatore indicato dall'indice passato in input, solitamente fornito da *getActionIndex*;
- **getX** e **getY**: servono a recuperare le coordinate del singolo puntatore indicato dall'indice, anche in questo caso, ottenuto con *getActionIndex*;
- **getActionMasked**: ritorna il codice identificativo del tipo di

azione compiuta.

Per quanto riguarda gli **eventi**, nel capitolo precedente, avevamo attuato un'analisi del click su schermo suddividendolo nelle fasi di *ACTION_DOWN* (posare il dito sullo schermo) e *ACTION_UP* (sollevare il dito). Nel multitouch, la filiera delle fasi da osservare sarà leggermente più lunga:

- *ACTION_DOWN*: rappresenta l'inizio del movimento. Corrisponde al primo puntatore che tocca lo schermo. Le informazioni

corrispondenti si troveranno all'indice 0 del MotionEvent;

- *ACTION_POINTER_D* per ogni altro puntatore, dopo il primo, che toccano lo schermo nella stessa "gestualità". Il metodo *getActionIndex()* fornirà la posizione nel MotionEvent che contiene i dati di questo singolo evento;
- *ACTION_MOVE*: non vengono aggiunti puntatori ma avviene un cambiamento, come uno spostamento;

- *ACTION_POINTER_U* segnala che un puntatore non primario, ossia uno che ha toccato il display dopo il primo, viene sollevato;
- *ACTION_UP*: invocato quando anche il primo puntatore, quello che ha dato inizio al movimento, è stato sollevato. La gestualità si è completata.

Capitolo 41 – Rilevare le gesture

Abbiamo imparato finora a gestire il touch. Il display percepisce il modo in cui puntatori – in genere le dita dell'utente – ne toccano la superficie prendendo nota delle coordinate, dei movimenti e di tutte le variazioni relative.

L'insieme delle variazioni impresse dai puntatori viene intesa nel suo complesso come una gestualità, o **gesture** come più comunemente si usa dire.

In effetti, i touch e le relative modifiche potrebbero essere analizzate,

come visto, con oggetti *MotionEvent* all'interno del metodo di callback *OnTouchEvent*. Ma la frequenza con cui appaiono nel mondo mobile gestures comuni ha indotto sin da subito il team di Android a prevedere appositi metodi per la loro gestione.

Esiste nelle API Android una classe fondamentale per la gestione delle gestures, **GestureDetector**. Il suo costruttore prende in input due parametri, il Context ed un listener che implementa *OnGestureListener*:

```
public class MainActivity extends Activity implements OnGestureListener
{
    private GestureDetector detector;
```

```
@Override
        protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    detector=new GestureDetector(this, this);
}

@Override
  public boolean onTouchEvent(MotionEvent event)
{
    detector.onTouchEvent(event);
```

```
        return true;
    }
    ...
  ...
}
```

Lo stralcio di codice appena illustrato mostra come può essere impostata un'Activity in cui vengano trattate le gesture. In dettaglio, si è:

- implementata l'interfaccia *OnGestureListener* nell'Activity;

- impostato un *GestureDetector* come membro privato;
- inizializzato il detector all'interno dell'*onCreate* passando due riferimenti all'Activity stessa: il primo in quanto Context, il secondo come listener per le gestures;
- fatto override del metodo *onTouchEvent* dell'Activity che "gira" il MotionEvent ricevuto al detector. Questo passaggio è fondamentale: stabilisce l'aggancio tra la

ricezione dei touch e la loro interpretazione in quanto gestures.

L'implementazione di *OnGestureListener* richiede l'override di diversi **metodi che rappresentano le principali gestures**:

- **onDown**: rappresenta l'atto di posare il dito sul display. È praticamente l'inizio di qualunque gesture;
- **onLongPress**: viene invocato solo quando la pressione sul display

rimane prolungata;

- **onFling**: è ciò che comunemente viene chiamato swipe, l'atto del dito con cui si mima lo sfogliare delle pagine verso destra o sinistra;
- **onScroll**: richiama il concetto comune di fare scorrere una lista di elementi verticalmente o orizzontalmente. Come gesture, viene associato in senso più ampio a qualsiasi sfregamento del dito sul display senza distacco;

- **onShowPress**: dovuto alla pressione del dito sul display nello stesso punto che non è ancora stata interrotta quindi al momento non interpretabile in altra maniera;
- **onSingleTapUp**: il classico tap, un colpetto singolo su display assimilabile ad un click.

Come si può prevedere, un'unica gesture può richiamare più metodi tra quelli citati. Ad esempio, pensiamo al fling, la sequenza di metodi che viene invocata è:

- *onDown* al momento di posare il dito sul display;
- *onScroll* – non uno solo ma una serie – durante lo sfregamento del dito sul display;
- *onFling*, al finale, solo quando il dito viene distaccato dal display.

Solo l'ultimo punto rappresenta il completamento della gesture con la sua definitiva connotazione.

Visto che ne abbiamo fatto cenno vediamo un piccolo esempio che può

essere attuato all'interno del metodo *onFling*:

```
@Override
public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY)
{
  if (e1.getX()>e2.getX())
        Toast.makeText(getApplicationContext(), "verso sx", Toast.LENGTH_SHORT).show();
  else
        Toast.makeText(getApplicationContext(), "verso dx", Toast.LENGTH_SHORT).show();
  return true;
}
```

questa implementazione mostra che vengono ricevuti due oggetti *MotionEvent*, uno rappresenta il punto di inizio della gesture (il momento di down in cui il dito è stato posato sul display) e l'altro che rappresenta il punto in cui il dito viene sollevato completando il fling.

La differenza tra le ascisse dei due oggetti *MotionEvent* permette di scoprire se lo swipe è stato eseguito verso destra o sinistra, risultato che viene notificato mediante Toast.

Capitolo 42 – Bluetooth

Auricolari, tastiere, mouse. Ma anche modem, navigatori e vivavoce. Il termine **Bluetooth** richiama alla mente una serie di dispositivi diffusissimi nel quotidiano dell'utente. In generale, stiamo parlando di un protocollo per la comunicazione wireless a brevi distanze, solitamente operante nel raggio di alcuni metri. Gli smartphone attuali lo integrano nella maggior parte dei casi e non potrebbe essere diversamente date le potenzialità di interazione a livello elettronico che offre.

Inoltre se da un lato molti dispositivi che fino a pochi anni fa venivano utilizzati in Bluetooth sono ormai stati

assorbiti – a livello di funzionalità – dagli smartphone (si pensi al navigatore satellitare) l'attuale mercato della microelettronica offre nuovi strumenti altamente configurabili per i quali questa via di comunicazione appare uno scenario efficiente e produttivo. In questa categoria rientra la famosissima scheda programmabile **Arduino** o il microcomputer **Raspberry Pi**. Possono essere impiegati in soluzioni utili in ambienti domestici, lavorativi o industriali ed un'app Android sarebbe la soluzione ideale per attuarne il controllo.

Sono quindi tanti i motivi per affrontare l'integrazione di Bluetooth nelle API Android ed il presente

capitolo mostra una panoramica dei principali concetti utili per mettersi in marcia.

Discovery dei dispositivi

La prima fase, solitamente necessaria, per lavorare con Bluettoth è la ricerca (anche denominata discovery, scoperta) d i **dispositivi disponibili** nei paraggi. L'esempio che stiamo per mostrare si occupa proprio di questo. Cercherà eventuali controparti nei paraggi e ne mostrerà i nomi in una *ListView*.

Per fare ciò è necessario innanzitutto che:

- il nostro dispositivo Android disponga del supporto Bluetooth;
- nell'*AndroidManifest*

della nostra applicazione siano predisposte le necessarie permission:

<uses-permission android:name="android.permission.BLUETOOTH"

*<uses-permission android:name="android.permission.BLUETOOTH_*

Oltre che essere disponibile nel dispositivo, **il Bluetooth dovrebbe essere attivato**. Ciò non è stato richiesto tra i prerequisiti visto che nell'esempio seguente impareremo ad affrontare anche questa situazione.

Il layout dell'app è molto semplice, contiene solo un pulsante per attivare la ricerca ed una *ListView* per mostrarne i

risultati.

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/re
    xmlns:tools="http://schemas.android.com/tool
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Cerca dispositivi"
        android:onClick="scan"/>

                                        <ListView
```

```
android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/listview"/>

</LinearLayout>
```

All'interno dell'*Activity* Java ci occuperemo, come presumibile, di istanziare tutti gli oggetti necessari al funzionamento del layout, in primis *Adapter* e *ListView*, ma soprattutto di approntare il necessario all'utilizzo di Bluetooth.

La classe prioritaria per l'integrazione è ancora una volta uno specifico Adapter: **BluetoothAdapter**.

Sarà lui, una volta che il supporto per il protocollo sarà correttamente attivato, a cercare "in giro" eventuali dispositivi con cui comunicare.

L'impostazione dell'Activity è la seguente:

```
public class MainActivity extends Activity
{
    private BluetoothAdapter btAdapter;
    private Set<bluetoothdevice> dispositivi;
    private ListView lv;
        private ArrayAdapter<string> adapter = null;
                    private static final int BLUETOOTH_ON=1000;
    . . .
```

. . .

}

</string></bluetoothdevice>

La costante intera *BLUETOOTH_ON* servirà solo come codice di richiesta nell'uso dell'Intent. Nelle righe precedenti si vede anche un'altra classe legata al mondo Bluetooth ed è **BluetoothDevice**. Il Set di oggetti di questo tipo conterrà i riferimenti dei dispositivi individuati nelle vicinanze.

Il metodo *onCreate* predispone le configurazioni iniziali:

@Override

protected void onCreate(Bundle

```
savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
                                    btAdapter =
BluetoothAdapter.getDefaultAdapter();
        lv = (ListView)findViewById(R.id.listview);
            adapter=new  ArrayAdapter<string>
(this,android.R.layout.simple_list_item_1);
        lv.setAdapter(adapter);
    }
</string>
```

mentre il metodo *scan* serve a gestire il click sul pulsante:

```
public void scan(View v)
{
    if (!btAdapter.isEnabled())
    {
                    Intent turnOn = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENAE
                    startActivityForResult(turnOn,
BLUETOOTH_ON);
    }
    else
        load();
}
```

Qui iniziamo ad entrare nel vivo delle funzionalità. Infatti, come detto in

precedenza, il supporto Bluetooth nel dispositivo Android deve essere presente ed attivato. Il metodo *isEnabled* cercherà di verificare proprio questo ed in caso di esito negativo lancerà l'Intent che chiederà all'utente il permesso di farlo (vedere in figura).

La finestra di dialogo sarà proprio il

risultato dell'Intent lanciato. Visto che si è usato il metodo *startActivityForResult*, il risultato verrà consegnato nel metodo *onActivityResult* di cui faremo l'override:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
        super.onActivityResult(requestCode, resultCode, data);
    if (requestCode==BLUETOOTH_ON && resultCode==RESULT_OK)
    {
        load();
    }
```

```
}
```

Come si può vedere per essere sicuri che l'utente ha accettato l'attivazione del Bluetooth è necessario che il codice di richiesta sia pari a *BLUETOOTH_ON* ed il codice di ritorno a *RESULT_OK*, una costante definita all'interno della classe Activity.

Infine il metodo load, più volte invocato, contiene il codice necessario per reperire i dispositivi ed inserirne i nomi all'interno dell'adapter:

```
private void load()

{

    dispositivi = btAdapter.getBondedDevices();
```

```
        adapter.clear();
        for(BluetoothDevice bt : dispositivi)
            adapter.add(bt.getName());
    }
```

La figura seguente mostra i nomi dei dispositivi trovati. Nel caso dell'esempio è stato individuata una sola interfaccia Bluetooth con cui comunicare il cui nome è *ospite-pc-0*.

Capitolo 43 – Scattare una foto

Potrebbe essere utile integrare nelle proprie app la capacità di scattare direttamente una foto ed utilizzarla "al volo". Per farlo sfrutteremo le potenzialità già incluse nel sistema per la consueta logica di "non inventare di nuovo la ruota".

Il dispositivo non mette solo a disposizione la macchina fotografica in quanto componente hardware ma anche il software e le API di gestione. Scopo di questo capitolo sarà imparare ad aprire dall'app direttamente il programma per scattare la foto ed ottenere come risultato l'immagine

acquisita. Il vantaggio di integrare il programma ufficiale per le foto sta nell'avere a disposizione tutta la sua completezza: gestione di zoom, effetti, configurazioni.

Il primo concetto da affrontare è l'**uso degli Intent per attivare il software fotografico**. Non stiamo affrontando un argomento nuovo, dal momento che gli Intent li abbiamo già usati per invocare l'apertura di un'Activity secondaria in un'app. Avevamo, in quell'occasione, preannunciato l'importanza di questo meccanismo e la sua capacità di recapitare messaggi al di fuori dell'applicazione, a livello di sistema operativo. Adesso è giunto il momento di vederlo in pratica.

L'Intent che useremo richiede l'esecuzione di un'azione a livello globale, definita mediante una costante presente in un ContentProvider di sistema, il MediaStore:

Intent photoIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);

startActivityForResult(photoIntent, PHOTO_REQUEST_CODE);

Il metodo utilizzato per avviare l'Intent è **startActivityForResult**. Si tratta di un meccanismo che permette di ricevere un risultato nella Activity chiamante. Il valore *PHOTO_REQUEST_CODE* è una costante intera definita nell'Activity stessa e serve solo a fornire un ID della

richiesta.

Il risultato – quindi il completamento della fotografia – sarà fruibile all'interno del metodo *onActivityResult* e sarà reperibile mediante il campo *data* degli Extras:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
        super.onActivityResult(requestCode, resultCode, data);
        if (requestCode==PHOTO_REQUEST_CODE)
        {
                Bitmap bp = (Bitmap)
```

```
data.getExtras().get("data");
            photo.setImageBitmap(bp);
        }
    }
```

Il codice per la gestione dell'evento non fa altro che prelevare l'oggetto Bitmap corrispondente alla foto scattata e posizionarlo all'interno di una ImageView, qui rappresentata dalla variabile *photo*. Ciò permetterà di vedere la versione rimpicciolita della foto, innestata all'interno del layout.

Capitolo 44 – Registrare un video

Dopo aver imparato a scattare una foto ed importarla nel progetto come Bitmap, è arrivato il momento di sfruttare la caratteristica "gemella": la **registrazione di un video**.

L'approccio è il medesimo del capitolo precedente ed i concetti di punta sono gli stessi. Li riassumiamo:

- l'attivazione del software di sistema per la realizzazione di video viene invocato mediante un action impostata

all'interno del MediaStore, un ContentProvider di sistema;

- l'Intent per il lancio dell'action sarà inoltrato con il metodo *startActivityForResult*;
- dopo la registrazione del video, verrà eseguito il metodo *onActivityResult*, grazie al quale potremo sfruttare i risultati.

Le righe seguenti inviano l'Intent per l'apertura del software della videocamera di sistema:

Intent videoIntent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);

startActivityForResult(videoIntent, VIDEO_REQUEST_CODE);

Queste istruzioni possono essere inserite in qualunque punto dell'Activity. Tipicamente verranno collocate in un metodo che gestisce, ad esempio, il click di un pulsante. La costante *VIDEO_REQUEST_CODE* è di tipo *int* ed è stata definita nell'Activity per riconoscere con un ID l'invocazione.

Il codice che segue mostra il metodo *onActivityResult*, utile per gestire il ritorno del controllo all'Activity, subito dopo aver girato il video.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
                super.onActivityResult(requestCode, resultCode, data);
                if (VIDEO_REQUEST_CODE == requestCode)
    {
        Uri videoUri = data.getData();
        video.setVideoURI(videoUri);
        video.start();
    }
}
```

Nell'esempio precedente si vede come è stato utilizzato il codice di richiesta *VIDEO_REQUEST_CODE*, e chiaramente a questo punto si potrebbe svolgere qualunque operazione. In questo caso specifico si è scelto di avviare direttamente la riproduzione del video.

Si noti che è stato utilizzato un controllo **VideoView**, una tipologia di View che permette di riprodurre un video di cui si è passato un riferimento tramite il metodo *setVideoURI*. Tale riferimento viene recuperato, sotto forma di URI, dai dati di ritorno dell'Intent.

Capitolo 45 – MediaPlayer: riprodurre file audio

La **multimedialità** è una delle facoltà più importanti di un dispositivo mobile. Basti pensare a quanto siano state innovative e di successo le introduzioni della radio e del lettore MP3 sui telefoni cellulari. Anche Android, quindi, ha un suo componente integrato per la fruizione di contenuti multimediali: il **MediaPlayer**.

In questo capitolo si imparerà ad utilizzarlo integrandolo rapidamente all'interno di un'app. Con poche righe di codice **creeremo un lettore MP3**

rudimentale ma funzionante, in grado di riprodurre una sola canzone, che per sempicità salveremo tra le risorse con ID *R.id.canzone*. Partendo dall'esempio che vedremo, estenderne l'utilizzo a raccolte di file musicali risulterà abbastanza semplice.

I metodi che sfrutteremo della classe MediaPlayer sono i seguenti:

- **create**: serve per ottenere un riferimento all'istanza del MediaPlayer. Non ne creeremo direttamente uno nuovo mediante l'operatore *new* di Java,

ma faremo in modo che sia il sistema stesso ad instanziarlo. Tra i parametri di questo metodo passeremo anche l'ID della risorsa che corrisponde alla canzone da ascoltare;

- **play**, **stop** e **pause**: dopo tutti i mangianastri, videoregistratori e lettori CD che hanno affollato la nostra vita sin da piccoli, appare quasi superfluo spiegare a cosa servano questi tre metodi. A scanso di equivoci, specifichiamo che essi

consentono rispettivamente di avviare, arrestare e mettere in pausa la fruizione del contenuto multimediale;

- **getDuration** e **getCurrentPosition**: forniscono informazioni temporali sull'esecuzione del contenuto. Il primo restituisce la durata complessiva della canzone mentre con il secondo otteniamo l'intervallo tempo che separa l'inizio della canzone con la posizione

temporale attualmente in riproduzione.

Il lettore MP3

Nel layout del nostro lettore inseriremo una SeekBar (che mostrerà l'avanzamento della musica in esecuzione) ed i pulsanti che consentono di utilizzare i controlli fondamentali:

```
<TableLayout
xmlns:android="http://schemas.android.com/apk/re

    android:layout_width="match_parent"

    android:layout_height="match_parent">

    <TableRow

        android:gravity="center"

        android:padding="20dp">

        <SeekBar

            android:id="@+id/bar"
```

```
            android:layout_width="0dp"
            android:layout_weight="1"
            android:layout_height="wrap_content"
/>
        </TableRow>
        <TableRow
            android:gravity="center"
            android:padding="20dp">
            <Button
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:onClick="play"
                android:text="Play" />
            <Button
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="pause"
        android:text="Pause" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="stop"
        android:text="Stop" />
  </TableRow>
</TableLayout>
```

Il risultato finale è mostrato nella figura seguente.

All'interno dell'Activity, il metodo *onCreate* fisserà i riferimenti principali alla SeekBar e al MediaPlayer:

```
public class MainActivity extends Activity

{


    private MediaPlayer mp=null;

    private Handler handler = new Handler();

    private double startTime = 0;
```

```
private SeekBar sk=null;


@Override
        protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    sk=(SeekBar) findViewById(R.id.bar);
                mp=MediaPlayer.create(this, R.raw.canzone );


}


. . .
```

```
    . . .
}
```

I metodi *play*, *pause* e *stop* sono molto semplici, a dimostrazione della facilità di interazione con il MediaPlayer:

```
private Runnable updateBar = new Runnable()
{
    public void run()
    {
      startTime = mp.getCurrentPosition();
      sk.setProgress((int)startTime);
      handler.postDelayed(this, 100);
    }
```

```
};

public void play(View v)
{
    mp.start();
    sk.setMax((int) mp.getDuration());
    handler.postDelayed(updateBar,100);
}

public void pause(View v)
{
    mp.pause();
}
```

```
public void stop(View v)
{
    mp.stop();
}
```

Nel codice presentato, si vede che per richiedere le tre operazioni di base del Mediaplayer – start, stop, pause – è sufficiente richiamare i metodi omonimi. Tra di essi si inserisce il meccanismo di **aggiornamento della barra di progresso**. Il suo avanzamento rappresenta l'andamento della riproduzione. Essa è realizzata in maniera piuttosto semplice: si è usato un handler temporizzato che legge ogni 100 millisecondi la posizione attuale di

riproduzione, ed attui il conseguente aggiornamento della barra. Il valore massimo cui il progresso può arrivare è la durata totale del brano, e viene impostato all'interno del metodo *play* utilizzando come fonte di informazione il risultato del metodo *getDuration* del Mediaplayer.

Capitolo 46 – MediaPlayer: riprodurre file video

Nel capitolo precedente abbiamo descritto la classe *MediaPlayer* come uno strumento in grado di fornire funzionalità multimediali, che consentono la fruizione di contenuti audio e **video** in un'app Android. Nell'esempio presentato in precedenza lo si è utilizzato per realizzare un semplice lettore MP3.

In questa lezione, sperimenteremo le sue capacità di gestione dei video. Grazie al MediaPlayer, infatti, potremo **riprodurre un filmato all'interno della**

nostra app. L'esempio che vedremo utilizzerà un file multimediale contenuto tra le risorse del progetto, ma accenneremo anche all'utilizzo del MediaPlayer per la **fruizione di filmati disponibili in rete**.

L’esempio

Il video che vorremo visualizzare sarà collocato nelle risorse, in particolare nella cartella *raw* destinata a contenere file di una tipologia non perfettamente inquadrabile tra le categorie standard (*layout*, *values*, *menu*, *drawable*, ecc…). Nel codice che verrà presentato, pertanto, ci riferiremo al video come risorsa, ed il suo ID sarà *R.raw.video*.

Quanto già appreso in precedenza in merito al MediaPlayer resta valido. In questo caso ci occuperemo per lo più di istanziarlo (sempre con il metodo di comodo `create`), avviare la proiezione del video e rilasciare le risorse al

termine delle operazioni.

Il layout

Si è volutamente scelto di utilizzare un layout molto semplice. L'unico widget al suo interno svolge il ruolo di "contenitore" del video:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/re

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:background="#EEFFCD"

    android:padding="20dp" >


    <SurfaceView

       android:id="@+id/surfView"

       android:layout_width="match_parent"
```

android:layout_height="300dp" />

</RelativeLayout>

Il widget, in particolare, è un controllo di classe *SurfaceView*. Si tratta di una superficie disegnabile "incastonata" nella finestra dell'interfaccia utente. Solitamente, non è necessario specificare altre impostazioni, se non la posizione e le dimensioni.

I contenuti al suo interno vengono manipolati mediante un oggetto di classe *SurfaceHolder*. Un riferimento ad esso può essere recuperato tramite il metodo `getHolder()`, disponibile nella classe

SurfaceView.

Il codice

L'Activity implementa l'interfaccia *SurfaceHolder.Callback*, che richiede il completamento di tre metodi: `surfaceCreated`, `surfaceChanged`, `surfaceDestroyed`. La loro invocazione si verificherà, rispettivamente, la prima volta che la superficie viene creata, ogni volta che subisce dei cambiamenti e quando viene distrutta.

Nel nostro esempio, implementiamo soltanto il metodo *surfaceCreated*. Al suo interno, sicuri dell'avvenuta creazione della superficie, potremo predisporre il codice di avvio del video.

Ecco l'Activity:

```
public class MyActivity extends Activity implements SurfaceHolder.Callback
{
    private MediaPlayer mediaPlayer;
    private SurfaceHolder holder;
    private SurfaceView surface;

    @Override
        protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_my);
                surface = (SurfaceView) findViewById(R.id.surfView);
```

```
        holder = surface.getHolder();
        holder.addCallback(this);
    }

    @Override
    public void surfaceCreated(SurfaceHolder surfaceHolder) {
                                                mediaPlayer=MediaPlayer.create(this,R.raw.video);
        mediaPlayer.setDisplay(holder);
        mediaPlayer.setOnPreparedListener(
                                                new MediaPlayer.OnPreparedListener() {
            @Override
                                                public void onPrepared(MediaPlayer mediaPlayer) {
```

```
                mediaPlayer.start();
            }
        }
    );

    mediaPlayer.setOnCompletionListener(new
MediaPlayer.OnCompletionListener() {
        @Override
        public void onCompletion(MediaPlayer
mediaPlayer) {
            mediaPlayer.release();
        }
    });
}
```

```
    @Override
     public void surfaceChanged(SurfaceHolder surfaceHolder, int i, int i2, int i3) {


    }


    @Override
    public void surfaceDestroyed(SurfaceHolder surfaceHolder) {


    }
}
```

Appena avviato l'esempio, vedremo il video apparire ed andare in esecuzione nel layout. Prima di farlo

però dovremo ricordarci di inserire un video nelle risorse.

All'interno del metodo `onCreate` dell'Activity, non facciamo altro che svolgere impostazioni di base: assegnamo un layout alla UI e facciamo in modo che il *SurfaceHolder* utilizzi l'Activity stessa come listener per i propri eventi di callback.

Più interessante è invece il codice all'interno del metodo `surfaceCreated`. Per prima cosa, viene recuperato un riferimento al MediaPlayer tramite `create()`, e viene passato anche l'ID del filmato come argomento. Successivamente assegnamo il display al MediaPlayer indicando, in pratica, quale sarà il contenitore del

video.

Affinchè il video possa essere avviato al momento opportuno, invocheremo il metodo *start()* all'interno di un listener di classe *OnPreparedListener*. Analogamente, vorremo poter liberare memoria al termine della proiezione; pertanto, all'interno di un listener di tipo *OnCompletionListener*, invocheremo il metodo `release()`, sempre appartenente alla classe *MediaPlayer*.

Video “remoti”

Prima di terminare, vediamo come potere **visualizzare un video remoto**. In questo caso, il file che vogliamo riprodurre non sarà contenuto nelle risorse, bensì disponibile in rete ad un determinato URL. Per riprodurlo, quindi, dovremo:

- inserire nel file *AndroidManifest.xml* la permission per l’accesso a Internet:

```
<uses-permission
android:name="android.permission
/>
```

- invocare un overload del metodo `create()` che non richieda un ID di una risorsa, ma un oggetto *Uri* contenente l'indirizzo remoto del video:

mediaPlayer = MediaPlayer.create(this,Uri.parse("http://www.exar

Capitolo 47 – Gestire l'audio

Quello del multimedia è uno dei settori più floridi e di maggiore interesse delle applicazioni per dispositivi mobili. Abbiamo già visto come sia possibile utilizzare il MediaPlayer per la riproduzione di contenuti multimediali. In questo capitolo approfondiremo la tematica relativa alla gestione dell'audio. In particolare verranno trattati due aspetti: la **registrazione dell'audio** e l'**AudioManager**.

Registrare e riascoltare

L'esempio presentato è un grande classico del multimedia: una semplice applicazione con due pulsanti, uno per avviare/fermare la registrazione, l'altro per gestire l'ascolto.

Vediamo subito il layout, molto semplice:

```
<LinearLayout
xmlns:android="http://schemas.androic

    xmlns:tools="http://schemas.andro

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    >
```

```
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/btn_registra"
        android:text="Registra"
        android:onClick="registra"/>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/btn_ascolta"
        android:text="Ascolta"
        android:onClick="ascolta"/>

</LinearLayout>
```

l codice dell'Activity, oltre all'`onCreate()`, include diversi altri metodi che servono a gestire i pulsanti:

```
public class MainActivity extends Activity
{
    private MediaRecorder registratore = null;
    private MediaPlayer   riproduttore = null;

    private static String filename = null;

    private boolean ascoltando=false;
    private boolean registrando=false;
```

```
    @Override
    public void onCreate(Bundle args)
    {
        super.onCreate(args);
        setContentView(R.layout.activity_main);
                                        filename =
Environment.getExternalStorageDirectory().getAbs
        filename += "/registrazione.3gp";
    }

    @Override
    public void onPause() {
        super.onPause();
        if (registratore != null) {
```

```
            registratore.release();
            registratore = null;
        }

        if (riproduttore != null) {
            riproduttore.release();
            riproduttore = null;
        }
    }

    . . .
    . . .
}
```

I **membri privati** visibili nell'Activity sono:

- due booleani, chiamati *ascoltando* e *registrando*. Loro eventuali valori positivi indicherebbero che è in corso, rispettivamente, la riproduzione dell'audio e la registrazione;
- *filename* è una stringa che indica il nome del file in cui andremo a salvare l'audio registrato. Il salvataggio verrà fatto nella directory

principale dello storage esterno il che permetterà, in caso di dubbi, di cercarlo sulla SD card ed ascoltarlo con il lettore di sistema;

- i due protagonisti dell'esempio, *MediaRecorder* e *MediaPlayer*. Appartengono entrambi al package *android.media* e servono a registrare l'audio – il primo – e a gestirne la riproduzione, il secondo.

Il frammento di codice precedente mostra anche `onCreate` e `onPause`. Il primo metodo non fa altro che svolgere inizializzazioni, mentre il secondo fa un po' di pulizia in memoria controllando che il *MediaPlayer* ed il *MediaRecorder* siano non-nulli ed in tal caso chiede il rilascio delle rispettive risorse.

Al click su uno dei pulsanti del layout, verranno impartiti i **comandi per la registrazione e l'ascolto**. In entrambi i casi l'Activity cambierà nome al pulsante, assegnando un'etichetta appropriata. In tutto ciò, giocano un ruolo fondamentale i due booleani, denominati *ascoltando* e *registrando*, che permettono di avere sempre chiaro

in quale fase di lavoro l'applicazione si trovi. Questi i metodi che reagiscono al click dei pulsanti:

```
public void registra(View v)
{
    Button btn=(Button) v;
    if (registrando)
    {
        // serve ad interrompere
        fermaRegistrazione();
        btn.setText("Registra");
    }
    else
    {
```

```
        // serve ad iniziare la registrazione
        registra();
        btn.setText("Ferma registrazione");
    }
    registrando=!registrando;
}

public void ascolta(View v)
{
    Button btn=(Button) v;
    if (ascoltando)
    {
        // serve ad interrompere
        fermaRiproduzione();
```

```
            btn.setText("Ascolta");
        }
        else
        {
                // serve ad iniziare la riproduzione
dell'audio
            riproduci();
            btn.setText("Ferma");
        }
        ascoltando=!ascoltando;
    }
```

Infine, vediamo il cuore dell'esempio, la parte che conterrà il codice più significativo: i metodi che provvedono

ad avviare/interrompere riproduzione e registrazione.

```
private void riproduci() {
    riproduttore = new MediaPlayer();
    try
    {
        riproduttore.setDataSource(filename);
        riproduttore.prepare();
        riproduttore.start();
    }
    catch (IOException e)
    {
        // gestisci eccezione
    }
```

```
}

private void fermaRiproduzione() {
    riproduttore.release();
    riproduttore = null;
}

private void registra() {
    registratore = new MediaRecorder();
    registratore.setAudioSource(MediaRecorder.A
    registratore.setOutputFormat(MediaRecorder.
    registratore.setOutputFile(filename);
    registratore.setAudioEncoder(MediaRecorder
```

```
    try
    {
        registratore.prepare();
    }
    catch (IOException e)
    {
       // gestisci eccezione
    }

    registratore.start();
}

private void fermaRegistrazione() {
    registratore.stop();
```

```
        registratore.release();

        registratore = null;

    }
```

Per quanto riguarda il *MediaPlayer*, non abbiamo nulla da aggiungere a quanto visto nel capitolo al riguardo. Anche il *MediaRecorder* non riserva grandi sorprese. Come si può vedere, è uno strumento molto completo: non solo permette di impostare tutti i parametri della registrazione – tra cui il microfono come sorgente ed il formato *.3gp* – ma si occupa di tutta la fase di registrazione su file (è sufficiente impostargli nome e percorso della destinazione).

AudioManager

Oltre all'esempio riportato, è giusto citare un'altra classe appartenente al framework: **AudioManager**. Si tratta di un servizio di sistema, richiamabile quindi nel seguente modo:

AudioManager manager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);

Offre diversi metodi per configurare rapidamente vari aspetti dell'audio. Almeno due di essi meritano di essere citati per la grande importanza che rivestono:

- **setRingerMode**: permette di impostare il

volume della suoneria del dispositivo. Richiede in input un valore intero da scegliere tra valori costanti: *RINGER_MODE_SILENT* (silenzioso), *RINGER_MODE_NORMAL* (normale suoneria), *RINGER_MODE_VIBRATE* (con vibrazione);

- **adjustVolume**: permette di regolare il volume. La direzione della variazione di volume va descritta con apposite costanti: *ADJUST_LOWER* (per

diminuire) e *ADJUST_RAISE* (per aumentarlo).

Capitolo 48 – Android e la low latency

Dopo i capitoli tecnici che hanno mostrato le azioni più comuni che possono essere eseguite nel multimediale, ci soffermiamo, in questo capitolo, su una problematica particolare la cui risoluzione è tuttora in corso. Stiamo parlando della **latenza audio nei sistemi Android**. Tecnicamente, ci si riferisce alla latenza come al ritardo temporale che subisce un segnale audio quando passa attraverso un sistema. Questo "passare attraverso" comprende una sequenza di fasi che possono essere riassunte in tre diverse attività: **conversione analogico-**

digitale (fase di ingresso), **elaborazione**, **conversione digitale-analogico (fase di uscita)**.

Il pubblico interessato maggiormente a questa problematica è sicuramente quello dei professionisti della musica (DJ, musicisti o semplicemente appassionati), oltre ovviamente ad un settore piuttosto ampio di sviluppatori. Sul tema, non si può negare che il mondo Android si trovi in un certo ritardo rispetto ad Apple, anche se nelle ultime versioni è stato fatto molto per migliorare la latenza audio sui sistemi Android. La ricerca di una "cura" presenta difficoltà relative alla diversità delle origini del problema stesso: capacità hardware, librerie disponibili,

rapporto con il codice nativo, oltre alla consueta aggravante della frammentazione del panorama elettronico, che non rende universalmente valide le soluzioni individuate.

La corsa verso la **bassa latenza** ha accelerato molto il suo ritmo negli ultimi anni, vivendo un particolare momento di fama durante il Google IO 2013, in un talk riguardante il miglioramento delle performance audio:

La documentazione Android specifica che dalla versione 4.1 sono stati introdotti dei cambiamenti architetturali finalizzati alla soluzione del problema. Viene avvertito comunque il lettore che tutte le spiegazioni in merito fornite non

sono rivolte agli sviluppatori di applicazioni, ma ai produttori di hardware e relativi driver, per una corretta implementazione delle problematiche audio sulle nuove versioni del sistema.

Capitolo 49 – Animazioni con XML

In questa guida abbiamo già approcciato la sfera della grafica, quando abbiamo parlato di stili ed immagini. Questo capitolo vuole proporre l'approccio ad un settore della grafica molto vasto, affrontabile con soluzioni avanzate e pretenziose o in maniera più semplice ma comunque efficace. Inizieremo parlando delle **animazioni**, tipiche di presentazioni pubblicitarie e videogame, ma che possono essere presenti anche su altre tipologie di app in cui sia utile poter dare, per così dire, un tocco di vitalità.

Animazioni con XML

In precedenza, abbiamo scoperto che con i *Drawable* è possibile disegnare in **XML**. Ora scoprirermo che è anche possibile creare animazioni con questo formato di dati. Gli esempi che mostreremo sono probabilmente il modo più rapido per ottenere i primi risultati ed iniziare ad affrontare uno scenario così ampio con poco sforzo.

Le animazioni, come molte altre cose nei nostri progetti, sono **risorse**. La loro configurazione in XML dovrà essere inserita in un file all'interno della cartella *res/anim*.

Per gli esempi che utilizzeremo, predisporremo un semplice layout con

un testo “Hello world” in posizione centrale, ed un pulsante in alto a sinistra con su scritto “Attiva animazione”. Alla pressione di quest’ultimo controllo, l’animazione verrà avviata e sarà applicata alla scritta “Hello world”. Data la facilità di configurazione delle animazioni in XML, le direttive saranno inserite nel file *res/anim/animazione.xml* e sarà sufficiente sostituirne il contenuto per sperimentare nuove animazioni.

Il **layout** che utilizzeremo è il seguente:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/re

    xmlns:tools="http://schemas.android.com/tool
```

```
android:layout_width="match_parent"
android:layout_height="match_parent">

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentTop="true"
    android:layout_alignParentLeft="true"
    android:text="Avvia animazione"
    android:onClick="avvia"/>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

```
    android:layout_centerInParent="true"
    android:id="@+id/txt"
    android:text="@string/hello_world" />

</RelativeLayout>
```

Di seguito è mostrato, invece, il codice Java dell'Activity:

```
public class MainActivity extends Activity
{

    private Animation anim=null;
    private TextView txt=null;

    @Override
```

```
protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        txt=(TextView) findViewById(R.id.txt);
                                        anim =
AnimationUtils.loadAnimation(getApplicationContex
            R.anim.animazione);
    }

    public void avvia(View v)
    {
        txt.startAnimation(anim);
     }
```

```
}
```

Come si vede, è molto semplice, e somiglia agli esempi già visti. Le uniche novità sono:

- a classe **AnimationUtils** che, all'interno del metodo `onCreate`, carica l'animazione semplicemente richiamandone l'ID di risorsa;
- il metodo **startAnimation** che

attiva l'animazione sulla *View*, in questo caso una *TextView*.

Ciò che manca è **completare il file dell'animazione**, che abbiamo deciso di chiamare *animazione.xml*. Come primo esperimento realizzeremo una rotazione a 360 gradi, ripetuta 3 volte:

```
<?xml version="1.0" encoding="utf-8"?>
<set
xmlns:android="http://schemas.android.com/apk/re
    <rotate android:fromDegrees="0"
        android:toDegrees="2000"
        android:pivotX="50%"
        android:pivotY="50%"
```

```
    android:duration="360"
    android:repeatMode="restart"
    android:repeatCount="3"
    android:interpolator="@android:anim/cyc

</set>
```

I nuovi tag impiegati sono due: **<set&gt**; che racchiude l'insieme delle animazioni e **<rotate>**, specifico per l'operazione da effettuare. Ciò ci consentirà di eseguire l'animazione in seguito al click dell'apposito pulsante.

Fatto questo, potremo provare un effetto di ingrandimento con il tag **<scale>**, semplicemente sostituendo il

contenuto del file *animazione.xml*:

```
<scale
    xmlns:android="http://schemas.android.co
    android:duration="1000"
    android:fromXScale="1"
    android:fromYScale="1"
    android:toXScale="4"
    android:toYScale="4"
    android:pivotX="50%"
    android:pivotY="50%">
</scale>
```

Il risultato sarà l'ingrandimento del testo "Hello world" di un fatto 4 (cioè, 4

volte più grande).

Quelli visti, ovviamente, sono solo degli esempi. La documentazione ufficiale permetterà di trovare tante altre opportunità ed idee. Sottolineaiamo comunque alcuni spunti interessanti:

- il tag **<set>** non deve necessariamente contenere una singola animazione, anzi il suo stesso nome richiama il concetto dell'insieme e lascia intendere che al suo interno possono essere concentrate più direttive XML;

- impostando opportunamento gli attributi sarà possibile ottenere effetti finali molto diversi. Oltre a *<scale>* e a *<rotate>* ci sono altri tag che molto utili, come **<alpha>** che regola le variazioni di trasparenza permettendo di creare effetti di dissolvenza, e **<translate>** che permette di ordinare spostamenti delle *View*;
- se si vogliono effettuare **sequenze di animazioni**, ed eseguire

operazioni tra di esse, è possibile trasformare l'Activity o un altro oggetto in un listener per i relativi eventi. Ciò viene fatto implementando l'interfaccia **AnimationListener** e comunicandolo all'istanza *Animation* con il metodo `setAnimationListener`. Fatto questo, si inserirà il codice nei metodi di cui è obbligatorio fornire l'override: `onAnimationStart`,

onAnimationEnd e
onAnimationRestart.

Capitolo 50 – Supporto multirisoluzione

La diffusione di Android su **dispositivi molto eterogenei** ha fatto la fortuna di questo sistema operativo, permettendogli di dimostrare le sue doti di adattamento. Ma come sappiamo ha creato non pochi grattacapi agli sviluppatori. Uno degli aspetti più delicati, infatti, è la necessità di adattare il layout al display.

Nel capitolo relativo alle risorse, si è spiegato come l'unità di misura da preferire sia il **Density-Indipendent Pixel**, in sigla *dp*, che rappresenta un pixel non vincolato dalla densità del display. Proprio questo concetto di

densità, intesa come rapporto tra numero di pixel e dimensioni dello schermo, ha assunto un ruolo di primo piano, prevaricando in importanza sia la risoluzione che la misura dello schermo.

Il passo concettuale doveroso, a questo punto, è l'abbandono della visione "pixel-centrica" delle misure proprio perchè questa non tiene conto della densità di popolazione dei pixel nel display. Definire le misure in pixel per gli elementi dei layout comporterebbe una visualizzazione molto diversa tra schermi a bassa densità (dove le immagini verrebbero allargate) e quelli ad alta densità (dove le immagini risultarebbero

rimpicciolite). **La densità viene misurata in dpi** ed il livello di 160 dpi (densità media o *mdpi*) rappresenta il punto di equilibrio. A questo livello, un pixel equivale ad un dp.

Gli altri livelli comuni per le densità sono:

- **ldpi**, la bassa densità, impostato a 120 dpi;
- **mdpi**, 160 dpi, come detto è la baseline delle densità;
- **tvdpi**, pensata per le tv: solitamente impostata a 213 dpi;

- **hdpi**, densità alta: 240 dpi;
- **xhdpi**, densità ultra-alta: 320dpi.

Per comprendere meglio quanto detto, può essere utile considerare la **relazione tra pixel e dp**, che può essere sintetizzata matematicamente come segue:

px = dp(dpi/160)*

Ciò può essere letto affermando che il rapporto tra pixel e dp è pari a quello tra la densità del display ed il livello di parità 160 dpi.

Non è un caso quindi che, tra le risorse di default inserite nei progetti di

Eclipse e Android Studio, l'immagine del logo del robottino verde (denominata *ic_launcher.png*) sia replicata in più cartelle *drawable*, ogni volta con dimensioni differenti:

- in *drawable-mdpi*, misura 48X48 pixel;
- in *drawable-hdpi*, 72X72 pixel;
- in *drawable-xhdpi*, 96X96 pixel.

Più è alta la densità di pixel cui è destinata l'immagine, più grandi sono le sue dimensioni. Ciò per contrastare il naturale rimpicciolimento che

provocherebbe un maggior numero di pixel, a parità di dimensioni. Tutto ciò avviene nel rispetto delle proporzioni dettate dalla formula descritta poc'anzi.

Best practises

Affinchè le nostre applicazioni possano avere una migliore diffusione su dispositivi eterogenei, è fondamentale dotarle di **interfacce fluide**, e per questo è bene tenere a mente alcuni consigli:

- utilizzare il **più possibile i RelativeLayout**, che ha per sua natura si adatta ai display;
- nel dimensionamento di *View* e *Layout*, evitare di dichiarare

esplicitamente le misure facendo ricorso il più possibile ai valori **wrap_content** e **match_parent**;

- tutte le misure necessarie vanno indicate in **dp** (o in **sp** per i font) evitando qualsiasi altra unità di misura che abbia attinenza col mondo reale (pixel, millimetri, pollici, etc…);
- **diversificare le risorse** in base alle possibili configurazioni dei dispositivi,

predisponendo i nomi delle cartelle caratterizzati dagli appositi modificatori. Ciò vale per le densità, con le directory *drawable-ldpi*, *drawable-hdpi*, eccetera, ma anche per le diverse modalità di orientamento del display (*layout-land* per landscape o *layout-port* per portrait) e per le dimensioni del display (*layout-small* per schermi non superiori ai 3 pollici, ad esempio). La serie completa dei

modificatori è disponibile sulla documentazione ufficiale.

Capitolo 51 – Animazioni con ViewPropertyAnimator

Abbiamo visto che è possibile utilizzare l'XML per effettuare le animazioni. Oltre a questo tipo di possibilità, è necessario conoscere anche un altro strumento, ovvero la classe **ViewPropertyAnimator**. Si tratta di un oggetto relativamente giovane del framework, introdotto con Android 3.1 che permette di attuare animazioni con un approccio totalmente "Java", tutto configurato via codice.

Questa classe offre le animazioni più comuni, che possono essere invocate mediante appositi metodi:

- **rotazione** lungo entrambi gli assi o solo rispetto all'asse X o Y: metodi `rotation`, `rotationX`, `rotationY`;

- **ingrandimento o riduzione**, anche questo su uno o più assi, con `scale()`, `scaleX()` e `scaleY()`;
- **traslazione** in una o più direzioni con `translation()`, `translationX()` o `translationY`;
- **trasparenza**, che può

essere modificata con il metodo `alpha`.

Per ottenere un riferimento ad un oggetto *ViewPropertyAnimator* è necessario invocare il metodo **animate()** sulla View. Ad esempio, il comando:

```
txt.animate().rotationY(180);
```

consente di effettuare la rotazione della View *txt* rispetto all'asse Y di 180 gradi.

È disponibile anche un altro meccanismo che avevamo apprezzato nelle animazioni XML: l'**attivazione di codice** in conseguenza a determinati

eventi connessi dell'effetto grafico. È possibile utilizzare più approcci.

Il primo è utilizzando un *AnimationListener*: un oggetto può implementare questa interfaccia e definire l'override dei metodi astratti. *ViewPropertyAnimator* utilizza il metodo `setListener`, che riceve il riferimento all'oggetto configurato.

In alternativa, si possono usare metodi specifici di *ViewPropertyAnimator*: **withStartAction** e **withEndAction**. In entrambi i casi di deve passare come parametro un oggetto *Runnable*, che include il codice da eseguire al momento opportuno.

Le righe di codice seguente svolgono la medesima rotazione vista in precedenza, ma mostrano un messaggio *Toast* alla conclusione:

```
txt.animate().rotationY(180).withEndAction(

new Runnable()

{

    Override

    public void run()

    {

        Toast.makeText(getApplicationContext(),
"Rotazione finita",
Toast.LENGTH_SHORT).show();

    }

});
```

Gli altri metodi disponibili e sopra citati possono essere utilizzati in maniera analoga, ed un buon esercizio potrebbe essere quello di applicare tali metodi per riprodurre gli stessi esempi citati nel capitolo sulle animazioni XML, ovviamente ignorando i file XML ed inserendo le invocazioni a *ViewPropertyAnimator* all'interno del metodo `avvia`.

Capitolo 52 – Accelerazione hardware

Una delle tendenze degli ultimi anni vede i dispositivi mobile guadagnare una fetta ulteriore di mercato sempre crescente rispetto ai PC tradizionali. Una domanda che in molti si stanno ponendo riguarda, ad esempio, la capacità dei tablet di poter sostituire nelle case di molti utenti i computer portatili. Oggetto di questo salto generazionale potrebbe essere quel pubblico composto dai non tecnici, ma dai normali utilizzatori interessati a consultare Internet, alla comunicazione e al multimedia.

In questa "lotta" tra dispositivi

desktop e mobile, giocano un ruolo fondamentale i videogame e la soddisfazione che può dare la grafica 3D realizzata per un tablet piuttosto che per un PC. Le prestazioni, in questo senso, non dipendono solo dall'elaborazione software, ma anche dalla disponibilità di risorse hardware.

Gli strumenti che hanno accompagnato sinora l'evoluzione della grafica sono soprattutto:

- le librerie **OpenGL**, utilizzabile in binding da Java o in modalità nativa da NDK;
- il potenziamento

dell'infrastruttura di calcolo, con **GPU** sempre più potenti ed un crescente utilizzo di **processori multicore** sui dispositivi.

Il problema con l'hardware sta nel fatto che, in genere, i potenziamenti si traducono in un maggior dispendio energetico, aggravando le problematiche di alimentazione già critiche sui dispositivi mobile.

Questa breve panoramica mostra il quadro della situazione, che ha portato ad un'importante innovazione in Android 3.0: la redirezione dell'intero sottosistema di UI verso

l'**accelerazione hardware**.

A partire da questa versione del sistema operativo, quindi, tutte le operazioni di rendering su Canvas di una View possono sfruttare maggiormente le potenzialità dell'hardware del dispositivo, prima tra tutti la GPU. L'accelerazione hardware può essere attivata o disattivata dalla configurazione dell'applicazione, per poterla utilizzare quando sia effettivamente utile. In alcuni casi, infatti, la sua attivazione spropositata potrebbe comportare problemi inaspettati, soprattutto su operazioni in 2D.

All'interno dell'*AndroidManifest*, l'attributo XML che permette di

abilitare/disabilitare l'accelerazione hardware è **android:hardwareAccelerated** ed il suo valore è di tipo booleano. Un aspetto molto importante è che, per evitare come detto spiacevoli inconvenienti, questa funzionalità può essere **applicata a vari livelli**:

- Applicazione;
- Activity;
- Window;
- View.

Inoltre, le View sono dotate del metodo **isHardwareAccelerated**, che

permette di controllare a runtime se l’accelerazione hardware è attiva.

Capitolo 53 – Gestire gli sms

La comunicazione via SMS potrebbe apparire un po' old-style in questi tempi così "social". Eppure si tratta della forma di messaggistica più diffusa al mondo, utilizzata con disinvoltura da persone appartenenti ad ogni fascia di età e disponibile su ogni dispositivo che disponga di funzionalità telefoniche. Relativamente al mondo Android, l'utilità in campo professionale dell'interazione via SMS è ancora ampia. Per fare un esempio, esistono molti dispositivi per la **domotica**, come caldaie, sistemi antifurto, etc. che notificano lo stato dell'impianto o il

verificarsi di situazioni particolari via SMS. Ecco: in un caso del genere un'app Android potrebbe rimanere in attesa di tali messaggi, ed utilizzarli interagendo con l'utente.

In questo capitolo, si affronterà proprio **l'invio e la ricezione di SMS in un'app Android**.

BroadcastReceiver

Delle quattro componenti che costituiscono un'applicazione Android, finora ne abbiamo viste tre: *Activity*, *ContentProvider* e *Service*. Qui introdurremo la quarta: il **BroadcastReceiver**.

Si tratta di un oggetto che si registra presso il sistema operativo, per essere allertato non appena si verifica una determinata circostanza. L'attivazione avviene mediante il classico meccanismo degli *Intent*, includendo altri elementi già visti come il *PendingIntent* e gli *IntentFilters*.

Un *BroadcastReceiver* – utile in tantissime circostanze, non solo per gli

SMS – viene creato estendendo la classe omonima ed implementando il metodo `onReceive`. Tale metodo riceve in input due parametri: il *Context* per l'interazione con il sistema e un *Intent*. Quest'ultimo contiene tutte le informazioni riguardanti l'evento.

Affinchè funzioni, **il BroadcastReceiver deve essere registrato nel sistema** e ciò può essere fatto in due modi:

- via **XML**, nell'*AndroidManifest*, mediante il tag `<receiver>`;
- in **Java**, usando il

metodo `registerReceiver`.

In questo capitolo ci avvarremo di entrambe le modalità di registrazione.

Invio di SMS

Per prima cosa impareremo ad **inviare SMS da un'app**. Per fare ciò, utilizzeremo una classe di sistema denominata *SmsManager*, adoperandola per due operazioni significative:

- ci faremo resituire un riferimento all'*SmsManager*;
- invieremo per suo tramite un messaggio di testo, con il numero del destinatario ed il testo.

Da non dimenticare che è necessario **dichiarare l'apposita permission**:

```
<uses-permission
android:name="android.permission
```

Per completezza, queste due operazioni saranno integrate con l'uso di *BroadcastReceiver* per notificare il successo nell'invio.

String numero = "3301234567"; // di pura fantasia

String testo = "Ciao, come stai?";

SmsManager smsManager = SmsManager.getDefault();

PendingIntent inviato = PendingIntent.getBroadcast(getApplicationContext(0, new Intent("SMS_INVIATO"), 0);

PendingIntent consegnato =

```
PendingIntent.getBroadcast(getApplicationContext(
0, new Intent("SMS_CONSEGNATO"), 0);


    suInvio=new BroadcastReceiver()
    {
      @Override
       public void onReceive(Context arg0, Intent
arg1)
      {
       if (getResultCode()==Activity.RESULT_OK)
                Toast.makeText(arg0, "SMS inviato
correttamente", Toast.LENGTH_LONG).show();
       else
            Toast.makeText(arg0, "Errore in invio",
Toast.LENGTH_LONG).show();
```

```
    }
  };

  suConsegna=new BroadcastReceiver()
  {
    @Override
      public void onReceive(Context arg0, Intent
arg1)
    {
      if (getResultCode()==Activity.RESULT_OK)
          Toast.makeText(arg0, "SMS consegnato",
Toast.LENGTH_LONG).show();
      else
                Toast.makeText(arg0,    "Errore",
Toast.LENGTH_LONG).show();
```

```
    }
  };

  registerReceiver(suInvio, new IntentFilter("SMS_INVIATO"));
  registerReceiver(suConsegna, new IntentFilter("SMS_CONSEGNATO"));

  smsManager.sendTextMessage(numero, null, testo, inviato, consegnato);
```

Il codice proposto è una modalità completa per gestire l'invio di un SMS. Il numero del destinatario ed il testo sono contenuti in due stringhe, che abbiamo chiamato rispettivamente `numero` e `testo`. Le due righe di codice

essenziali sono:

SmsManager smsManager = SmsManager.getDefault();

smsManager.sendTextMessage(numero, null, testo, inviato, consegnato);

La prima riga individua l'*SmsManager* e la seconda si occupa dell'invio. Tutto ciò che si trova tra loro serve a gestire la notifica di invio e consegna.

Ognuna di queste due fasi viene gestita con:

- un *PendingIntent* che congela un *Intent* relativo ad azioni

personalizzate (`SMS_INVIATO` per l'invio e `SMS_CONSEGNATO` per la consegna), che sarà inviato al momento del completamento di ognuna delle due fasi;

- un *BroadcastReceiver* che si registrerà per essere informato del lancio del *PendingIntent* corrispondente.

Nell'Activity sono stati definiti due membri di classe *BroadcastReceiver*:

private BroadcastReceiver suInvio = null;

private BroadcastReceiver suConsegna = null;

La loro inizializzazione vera e propria è stata effettuata in fase di invio. Il metodo `onReceive` in questo caso apre un Toast di notifica.

Affinchè il *BroadcastReceiver* sia attivo è necessario che venga registrato. Visto che ci sono due coppie *BroadcastReceiver-PendingIntent*, avverranno due registrazioni, ognuna delle quali assocerà un Receiver con il corrispondente IntentFilter relativo all'azione richiesta:

registerReceiver(suInvio, new IntentFilter("SMS_INVIATO"));

registerReceiver(suConsegna, new

IntentFilter("SMS_CONSEGNATO"));

Infine possiamo richiedere che nel metodo `onPause` vengano **cancellate le registrazioni dei BroadcastReceiver**, nel seguente modo:

```
@Override
protected void onPause()
{
  super.onPause();
  unregisterReceiver(suInvio);
  unregisterReceiver(suConsegna);
}
```

Ricezione

Per quanto riguarda la ricezione di SMS, non aggiungeremo concetti nuovi, dal momento che utilizzeremo nuovamente la classe *BroadcastReceiver*

Prima di tutto, **specifichiamo le permission nel Manifest**; questa volta occorrono le seguenti:

```
<uses-permission
android:name="android.permission.RECEIVE_SMS
```

```
<uses-permission
android:name="android.permission.READ_SMS"
/>
```

Poi creiamo una classe Java che eredita da *BroadcastReceiver*:

```
public class IncomingSMS extends BroadcastReceiver
    {
        @Override
        public void onReceive(Context context, Intent intent)
        {
            Bundle extras = intent.getExtras();


            if ( extras != null )
            {
                        Object[] smsExtra = (Object[]) extras.get( "pdus");
                for ( int i = 0; i < smsExtra.length; ++i )
```

```
            {
                SmsMessage sms =
SmsMessage.createFromPdu((byte[])smsExtra[i]);
                String testo =
sms.getMessageBody().toString();
                String numero =
sms.getOriginatingAddress();
                /* Svolgimento di operazioni
                 sul messaggio*/
            }
        }
    }
}
```

Infine registriamo presso il sistema il

BroadcastReceiver, inserendo un apposito tag nel manifest:

```
<receiver android:name=".IncomingSMS">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_
    />
    </intent-filter>
</receiver>
```

L'azione inclusa nell'*IntentFilter*, questa volta, non è definita da noi come nel caso dell'invio. Rimaniamo in attesa dell'evento *SMS_RECEIVED* definito nel package *Telephony*.

Per quanto riguarda più in dettaglio il

codice scritto in Java, tra gli *Extras* dell'*Intent* ce n'è uno con etichetta *pdus*. Dobbiamo trasformare le varie *PDU* arrivate – in pratica gli SMS grezzi - in oggetti di classe *SmsMessage*. Questo permetterà anche di poterli leggere in maniera agevole con metodi ad hoc. Dopo la conversione, infatti, per **ottenere il testo del messaggio** è sufficiente invocare il metodo `getMessageBody`, mentre per **il numero del mittente** si ricorre a `getOriginatingAddress`.

Capitolo 54 – Elenco delle chiamate

Dopo aver trattato l'invio e la ricezione degli SMS, possiamo dedicarci all'altra "metà del cielo" in fatto di telefonia: le **chiamate**. In questo capitolo, non affronteremo ancora l'inoltro e la gestione delle chiamate in arrivo, ma discuteremo l'**utilizzo dell'elenco storico delle telefonate inoltrate, ricevute e perse**.

Come viene gestito questo insieme di informazioni in Android? La discussione che segue si basa su concetti di persistenza già trattati, ed è emblematica in quanto questo tipo di gestione viene adottato anche per altre tipologie di

informazioni.

Abbiamo conosciuto l'utilizzo di **Sqlite** come database naturale per Android e successivamente abbiamo apprezzato le particolarità dei *ContentProvider* come sistema per condividere dati nel sistema. Inoltre si era già detto che Android fa uso di *ContentProvider* per gestire l'inserimento e la fruizione di insiemi di dati come i contatti, il calendario o il MediaStore.

Anche **i dati delle chiamate vengono fruiti mediante un ContentProvider**, ovvero **CallLog**.

Come abbiamo già sperimentato, quando ci si vuole interfacciare con un

ContentProvider di sistema, il principale problema è imparare a conoscerne la struttura, studiarne le classi e gli Uri disponibili. La buona notizia è che questo è praticamente l'unico problema, perchè l'**interfacciamento con i ContentProvider** avviene sempre alla stessa maniera.

L'Uri di riferimento – l'indirizzo cui fa capo questo sottosistema informativo – è *CallLog.Call.CONTENT_URI*. Quindi per recuperare un elenco delle chiamate è sufficiente eseguire questa richiesta:

```
Cursor
crs=getContentResolver().query(CallLog.Calls.CON
null, null, null, null);
```

Il *Cursor* ottenuto conterrà moltissimi record, ognuno dei quali si riferisce ad una chiamata effettuata, ricevuta o persa. Gli aspetti che prenderemo in considerazione sono identificati dai seguenti campi:

- **CallLog.Calls.NUMBI**
  il numero di telefono;
- **CallLog.Calls.TYPE**:
  il tipo di telefonata (inoltrata, ricevuta o persa). I possibili valori sono costanti: *CallLog.Calls.OUTGOIN* *CallLog.Calls.INCOMIN*

e *CallLog.Calls.MISSED_*

- **CallLog.Calls.DATE**: informazione temporale.

Nell'esempio che vedremo, ogni record elaborato sarà mostrato in un layout a tre colonne (file: *res/layout/callrow.xml*):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/re
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <TextView
```

```
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="2"
     android:maxLines="10"
    android:id="@+id/numero" />
<TextView
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:id="@+id/tipo" />
<TextView
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="2"
```

```
android:id="@+id/dataora" />

</LinearLayout>
```

Per il resto, l'esempio inserisce i dati raccolti all'interno di una *ListActivity*. Viene utilizzato un *CursorAdapter*, componente già visto, preparato appositamente per agganciare i risultati di una query al layout:

```
public class MainActivity extends ListActivity

{

 private CursorAdapter adapter=null;

    private SimpleDateFormat simple=new
SimpleDateFormat("dd/MM/yyyy HH:mm:ss");


 @Override
```

```
protected void onCreate(Bundle savedInstanceState)
{
 super.onCreate(savedInstanceState);
                                                Cursor crs=getContentResolver().query(CallLog.Calls.CON
null, null, null, null);

adapter=new CursorAdapter(this,crs,0)
{

  @Override
   public View newView(Context c, Cursor crs, ViewGroup vg)
   {
                                                View
```

```
v=LayoutInflater.from(c).inflate(R.layout.callrow,
null);

        return v;

    }


    @Override
      public void bindView(View v, Context ctx,
Cursor crs)

    {

                    TextView    numero=(TextView)
v.findViewById(R.id.numero);

        numero.setText(crs.getString(crs.getColumn

                    TextView    tipo=(TextView)
v.findViewById(R.id.tipo);

        switch(crs.getInt(crs.getColumnIndex(CallL

        {
```

```
                                                    case
CallLog.Calls.OUTGOING_TYPE:
                    tipo.setText("OUT");
                    break;

                case CallLog.Calls.INCOMING_TYPE:
                    tipo.setText("IN");
                    break;

                case CallLog.Calls.MISSED_TYPE:
                    tipo.setText("MISSED");
                    break;
            }
```

```
                    TextView   dataora=(TextView)
v.findViewById(R.id.dataora);
                    String   data=simple.format(new
Date(crs.getLong(crs.getColumnIndex(CallLog.Call

        dataora.setText(data);

    }

};


    setListAdapter(adapter);

 }


}
```

l risultato prodotto, mostrato nell'immagine seguente, è esattamente ciò che ci aspettavamo: l'elenco storico

delle chiamate.

Capitolo 55 – Gestire i contatti

La **rubrica** di Android è la sorgente principale di tutti i recapiti telefonici da poter contattare. Considerando che si tratta di un insieme di dati strutturati, disponibili a livello di sistema, non stupisce il fatto che essa sia stata implementata nei termini di un **ContentProvider**.

I **contatti**, ai quali normalmente ci si riferisce con il termine rubrica, rappresentano uno dei principali *ContentProvider* disponibili su Android. La sua struttura può sembrare piuttosto confusa, e per questo occorre innanzitutto fare un po' d'ordine.

Sul funzionamento dei *ContentProvider* non c'è nulla di nuovo da aggiungere rispetto a quanto visto nelle precedenti lezioni. In questa sede ricordiamo brevemente che un *ContentProvider* permette di accedere a dati condivisi tramite le classiche operazioni di creazione-lettura-modifica-cancellazione mediante i metodi *insert*, *query*, *update* e *delete*. A differenza di quanto accade con i database relazionali, essi non richiedono di accedere direttamente a tabelle, bensì di individuare la risorsa oggetto dell'operazione mediante un riferimento univoco di classe *Uri*.

Proprio dall'organizzazione delle classi e dal reperimento degli URI inizia

questo capitolo. L'**organizzazione dei contatti** viene distribuita su tre livelli:

- *Contact*;
- *RawContact*;
- *Data*.

Prima di spiegare le differenze tra i tre, si tenga a mente che per utilizzare un dispositivo Android, l'utente deve associare ad esso **uno o più account** Google. Il sistema operativo deve quindi prendersi carico di registrare i dati dei contatti ed accoppiarli con l'account che li utilizza.

Mentre i *Contact* rappresentano

singole persone indipendentemente dalla quantità di informazioni che di ognuna di esse si posseggono (email, telefono di casa, telefono dell'ufficio, cellulare, etc.), i *RawContact* includono l'associazione tra un contatto e un account. Inoltre per ogni persona esiste un solo *Contact* e per ogni *Contact* possono esistere più *RawContact*.

I *Data* invece rappresentano i singoli dettagli che formano ogni *RawContact*.

Le classi per la gestione dei contatti che useremo saranno pertanto tre:

- *ContactsContract.Con*
  i singoli contatti;

- *ContactsContract.Raw*
  i “raw contact”;
- *ContactsContract.Data*
  i dati di dettaglio.

Le permission

Consueto obbligo è quello di dichiarare le permission adeguate, dipendentemente dalle operazioni che si vogliono svolgere:

- per la sola **lettura dei contatti**, utilizzeremo la seguente sintassi:

<uses-permission
android:name="android.permission.READ_CONTA

- se vogliamo anche **inserire e modificare i dati**, includeremo quanto

segue:

<uses-permission

android:name="android.permission.WRITE_CONT

Lettura dei contatti

L'Uri per l'accesso all'insieme di dati (*Contact*, *RawContact* o *Data*) è contenuto in una costante stringa di nome *CONTENT_URI,* della classe *ContactsContract.Contacts*.

L'elenco dei contatti disponibili può essere letto tramite un *Cursor* con le seguenti righe di codice:

```
String[] projection=
{Contacts._ID,Contacts.DISPLAY_NAME};

Cursor crs=getContentResolver()

    .query(ContactsContract.Contacts.CONTENT_
projection, null, null, null);
```

In questo esempio, i dati inseriti nella proiezione sono solo due dei tanti disponibili, ma sono quelli più utili: l'ID del contatto, indispensabile per accedere a tutti gli altri dati, ed il nome del contatto.

Possiamo visualizzarli, come fatto in altre occasioni, combinando l'utilizzo delle classi *CursorAdapter* e *ListView*.

Inserimento di dati

Immaginiamo di voler inserire alcuni dati per un nuovo *RawContact*. Non dovremo fare altro che creare il nuovo contatto, ottenerne l'ID ed utilizzarlo per inserire singoli *Data*:

```
ArrayList<ContentProviderOperation>
operations = new
ArrayList<ContentProviderOperation>();

operations.add(ContentProviderOperation

        .newInsert(ContactsContract.RawContac

            .withValue(ContactsContract.RawCon
null)

            .withValue(ContactsContract.RawCon
null)

            .build());
```

```
    operations.add(ContentProviderOperation
            .newInsert(ContactsContract.Data.CONT
            .withValueBackReference(ContactsCo
0)
            .withValue(ContactsContract.Data.MI
                ContactsContract.CommonDataKin
            .withValue(ContactsContract.Common
“Guido Rossi”)
    .build());
        operations.add(ContentProviderOperation.
            .withValueBackReference(ContactsCo
0)
            .withValue(ContactsContract.Data.MI
                ContactsContract.CommonDataK
```

```
                .withValue(ContactsContract.Common
"0611223344")
                .withValue(ContactsContract.Common
ContactsContract.CommonDataKinds.Phone.TYPE_
                .build());
            operations.add(ContentProviderOperation.
                .withValueBackReference(ContactsCo
0)
                .withValue(ContactsContract.Data.MI
                    ContactsContract.CommonDataKi
                .withValue(ContactsContract.Common
"guido@rossi.it")
                .withValue(ContactsContract.Common
ContactsContract.CommonDataKinds.Email.TYPE_
                .build());
```

```
try {
    cr.applyBatch(ContactsContract.AUTHORITY
operations);
    }
catch (RemoteException e)
    {
    // gestione delle eccezioni
    }
catch (OperationApplicationException e)
    {
     // gestione delle eccezioni
    }
```

Quando operazioni di questo tipo vengono effettuate come un'aggregazione

di diversi inserimenti, può essere utile usare le **ContentProviderOperation**. I loro principali fattori di utilità sono l'uso di un builder per comporre in modo agevole la richiesta, e l'esecuzione dell'operazione in batch.

Nelle righe di codice precedenti per prima cosa viene creato un *ArrayList* di *ContentProviderOperation*. Poi vi si aggiungono una alla volta una serie di operazioni, utilizzando questi comandi:

- *newInsert* , per creare il nuovo inserimento;
- *withValue* , per specificare i parametri da inserire;

- *build* , per costruire il comando completo.

L’esecuzione vera e propria avverrà richiedendo al *ContentResolver* di eseguire il batch con il metodo *applyBatch*.

Capitolo 56 – AndroidManifest.xml e le capabilities

Abbiamo già discusso ed utilizzato il file *AndroidManifest.xml* durante questa guida. È chiaro che programmare applicazioni Android senza saperlo configurare è impossibile. L'uso che ne abbiamo fatto finora ha riguardato per lo più la definizione delle componenti da inserire nell'applicazione. Abbiamo visto che se si vogliono utilizzare le quattro componenti fondamentali – *Activity*, *ContentProvider*, *Service*, *BroadcastReceiver* – è necessario, oltre ad estendere l'opportuna classe Java, inserire un adeguato elemento XML nel

manifest, in particolare all'interno del nodo *<application>*:

```
<manifest
xmlns:android="http://schemas.android.com/apk/re
    package="..."
    android:versionCode="1"
    android:versionName="1.0" >
    . . .
    . . .
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
```

```
        <activity
            android:name="..."
            android:label="@string/app_name" >
            <intent-filter>
                <action
android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHE
/>
            </intent-filter>
        </activity>
        <receiver android:name="...">
            <intent-filter>
              <action android:name="..." />
            </intent-filter>
```

```
        </receiver>
        <provider
        android:name="..."
            android:authorities="..."/>
        <service android:name="..."/>
    </application>

</manifest>
```

In questo capitolo ci interesseranno altri particolari relativi al file manifest, ma che saranno inseriti esternamente al nodo `<application>`. Ci occuperemo di tutti quegli aspetti che descrivono i **requisiti** che un dispositivo deve possedere affinchè la nostra

applicazione possa esservi installata.

Si tratta di un preambolo indispensabile per pensare alla pubblicazione della nostra app.

L'elemento `<uses-sdk>` è molto importante per specificare il range di **versioni Android** in cui l'applicazione può funzionare. La finalità principale di questo nodo è garantire la retrocompatibilità dell'applicazione. Tramite questo elemento è possibile specificare l'API level, ovvero la versione delle API utilizzate, espressa con un numero intero. Non è necessario conoscere a memoria tutte (anche perchè l'elenco completo è sempre disponibile su Internet); tuttavia bisogna tenere presente che le "pietre miliari" della

storia di Android non sono molte. I principali API level sono i seguenti:

- 7 e 8: corrispondono ad Android 2.1 e 2.2 e costituiscono la soglia minima che ormai ha senso supportare nelle proprie applicazioni;
- 11: Android 3 (Honeycomb), che ha rappresentato una piccola rivoluzione, in cui sono state integrate alcune caratteristiche basilari nel framework (per esempio *Fragments* e

Loaders);

- 14: inizia Android 4 (Ice Cream Sandwich), l'alba della versione attuale del sistema operativo.

Gli attributi di `<uses-sdk>` sono:

- **minSdkVersion**: indica la minima versione supportata. Può andare bene, ormai, che sia 7 o 8 ma se è inferiore alla 11 sarà importante acquisire confidenza con la libreria

di supporto per integrare, nelle versioni più datate, le novità di Honeycomb. È importante che questo attributo sia esplicitamente dichiarato perchè, se non lo fosse, la minima versione supportata sarà l'API level 1 (e ciò, oggi come oggi, ha poco senso);

- **targetSdkVersion**: rappresenta la versione principale per la quale è stata pensata l'applicazione;
- **maxSdkVersion**:

molto poco utilizzato, anche perchè sconsigliato dalla stessa documentazione, questo attributo rappresenta il massimo API level supportato. Dal momento che lo scopo di `<uses-sdk>` è la gestione della retrocompatibilità, *maxSdkVersion* rischia di impedire l'installazione dell'applicazione su dispositivi più recenti.

Il tag `<permission>` è stato già incontrato parecchio nel corso di questa

guida. Necessario ogni volta che la nostra applicazione deve avviare comunicazioni o accessi particolari, richiede almeno che la definizione dell'attributo `android:name`, che specifica esattamente il tipo di permission richiesta. Finora l'abbiamo incontrato in:

- accesso alla Rete: `android.permission.IN`
- localizzazione: `android.permission.AC` e `android.permission.AC`

- storage esterno: `android.permission.WR`
- comunicazioni telefoniche: `android.permission.CA` e `android.permission.RE`
- *Content Providers*: vari casi (contatti, calendario, chiamate, etc…).

nfine, `<uses-feature>` è la risposta alla frammentazione hardware e software del sistema Android. Permette di specificare di quali **caratteristiche**

hardware o software l'applicazione ha bisogno per funzionare.

Gli attributi di cui dispone sono:

- `android:name`: una stringa che definisce quale dotazione del dispositivo è necessaria affinchè l'applicazione funzioni correttamente. A livello hardware, si potrebbe avere bisogno di verificare se l'equipaggiamento elettronico del device comprende tecnologie come Bluetooth o NFC,

nonchè dispositivi come la videocamera o lo schermo multitouch;

- `android:required`: è un valore booleano. Se impostato a `true` indica che la caratteristica è assolutamente obbligatoria per il funzionamento dell'app; altrimenti indica che la feature è fortemente consigliata ma non obbligatoria;
- `android:glEsVersion` indica la versione delle librerie OpenGL ES richiesta dall'app.

Capitolo 57 – Preparare l'app per la pubblicazione

Ora che abbiamo imparato i fondamenti dello sviluppo di applicativi Android rimane un ultimo passaggio per chiudere il cerchio: il **rilascio e la pubblicazione dell'app**. In questa lezione vedremo come preparare e "confezionare" un'applicazione Android per il rilascio nella principale area di pubblicazione per applicazioni Android: il Google Play Store.

Preparare ed eseguire il *packaging* di un'applicazione per la pubblicazione è un processo conosciuto come il nome di *release process*.

Il *release process* è il processo attraverso il quale passa la nostra app per diventare la *release build*, ovvero la versione ufficiale pronta per essere pubblicata. Prima di guadagnarsi questo appellativo, e quindi di essere disponibile per l'utente finale, l'applicazione è nella fase di *release candidate build*: solo dopo attente e rigorose verifiche e test l'app diventa una *release build* pronta per il rilascio.

Il processo per la pubblicazione di una applicazione Android si divide in 5 passi:

1. Preparare una *release candidate build* della

nostra applicazione.

2. Testare attentamente e minuziosamente la *release candidate*.
3. Generare un *package* Android e ufficializzarlo con una firma digitale.
4. Testare attentamente e minuziosamente il *package* pronto per il rilascio.
5. Pubblicare l'applicazione.

Prima di effettuare i test definitivi sulla nostra app dobbiamo preparare la *release candidate build*. Per fare questo

dobbiamo aver implementato e testato tutte le funzionalità utili al corretto funzionamento dell'applicazione, corretto tutti i bug e rimosso tutto il codice utilizzato per la diagnostica per evitare che incida nelle performance della app.

Per avere una *release candidate* ufficiale dobbiamo anche effettuare qualche modifica al file di configurazione *AndroidManifest.xml*. Alcune modifiche sono imposte dai marketplace come il *Google Play Store* mentre altre sono dettate dal buon senso e da linee guida comunemente accettate. Vediamo dunque come dobbiamo preparare l'*AndroidManifest.xml* per una app che vogliamo ufficializzare a

release candidate:

- Innazittutto verifichiamo che **l'icona dell'applicazione** sia impostata correttamente: questa icona sarà visualizzata agli utenti e verrà utilizzata dai marketplace per presentare l'app, quindi è importante che sia d'impatto e perfettamente configurata;
- Verifichiamo inoltre che il **nome dell'applicazione sia**

appropriato e correttamente impostato: questo sarà il nome con cui l'applicazione si presenterà agli utenti finali;

- Verifichiamo la corretta impostazione del testo indicante **la versione dell'applicazione**;
- Controlliamo che il **codice della versione dell'applicazione** sia impostato correttamente: questo è un codice che la piattaforma Android

utilizza per gestire gli aggiornamenti della nostra app;

- Verifichiamo **l'impostazione del** *uses-sdk* **dell'applicazione**: possiamo impostare il massimo, il minimo e il consigliato SDK Android supportato dall'applicazione. Il Google Play Store filtra le applicazioni disponibili per uno specifico utente in base alle informazioni fornite da ogni file di configurazione

AndroidManifest.xml, comprese quelle relative all'Android SDK;

- Verifichiamo di aver **disabilitato l'opzione** *debuggable*;
- Verifichiamo che tutti i **permessi richiesti dall'applicazione siano appropriati** per un corretto funzionamento della app: richiediamo solo i permessi davvero necessari e assicuriamoci di aver richiesto tutto il necessario indipendentemente da

come si potrebbe comportare il device senza di essi.

A questo punto possiamo passare alla **fase di testing**. In realtà questa fase non richiede particolari interventi, più che altro richiede particolari attenzioni e accorgimenti: dobbiamo testare la nostra applicazione il più rigorosamente possibile e verificare che l'app rispetti i criteri richiesti dal Google Play Store.

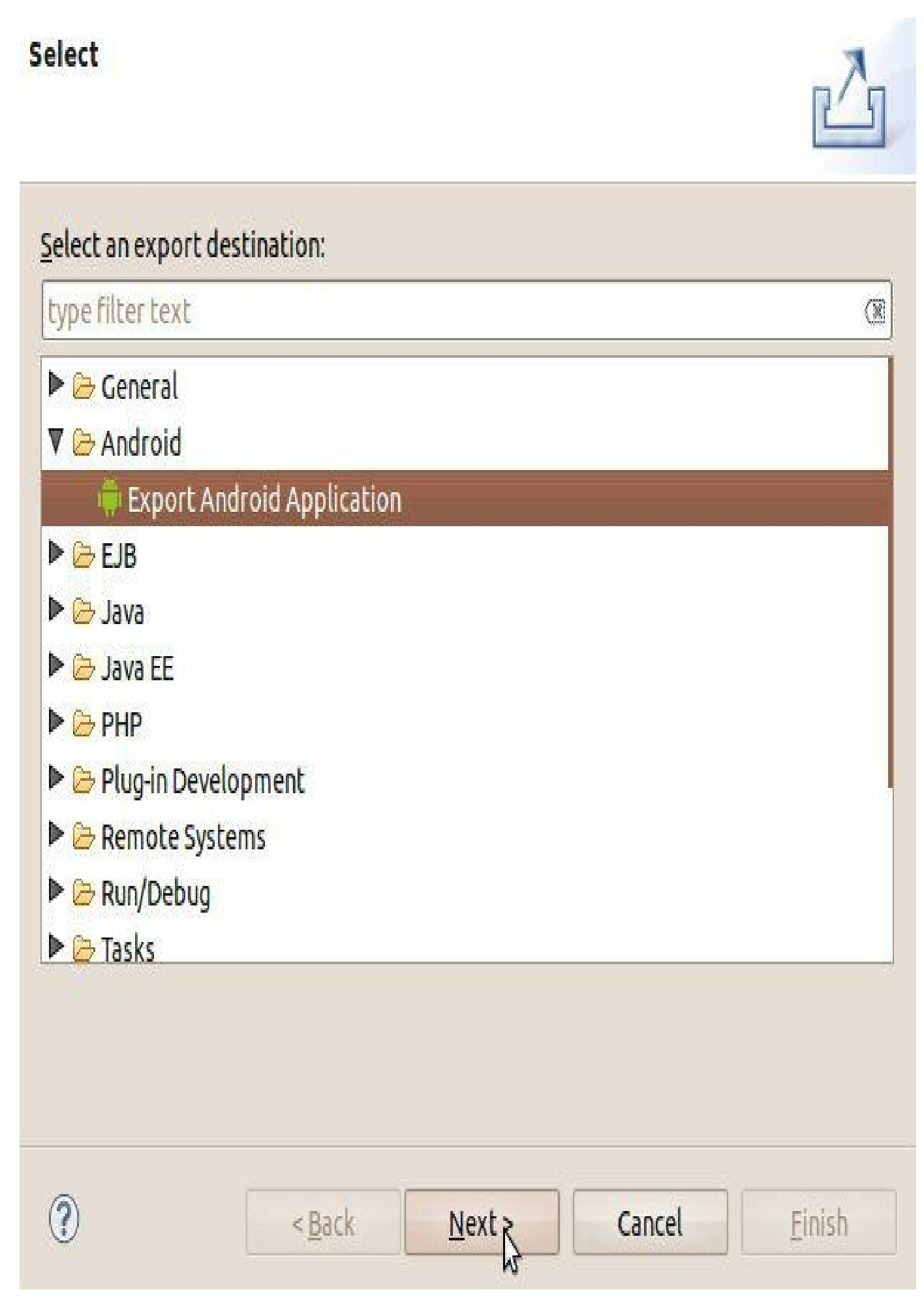
Se durante i test dovessimo riscontrare dei bug o delle problematiche di qualche genere sarà nostro compito valutare quanto siano importanti e serie, considerarne la gravità ed eventualmente valutare anche

la possibilità di **interrompere il processo di rilascio** per iniziarlo nuovamente una volta sistemate le problematiche riscontrate.

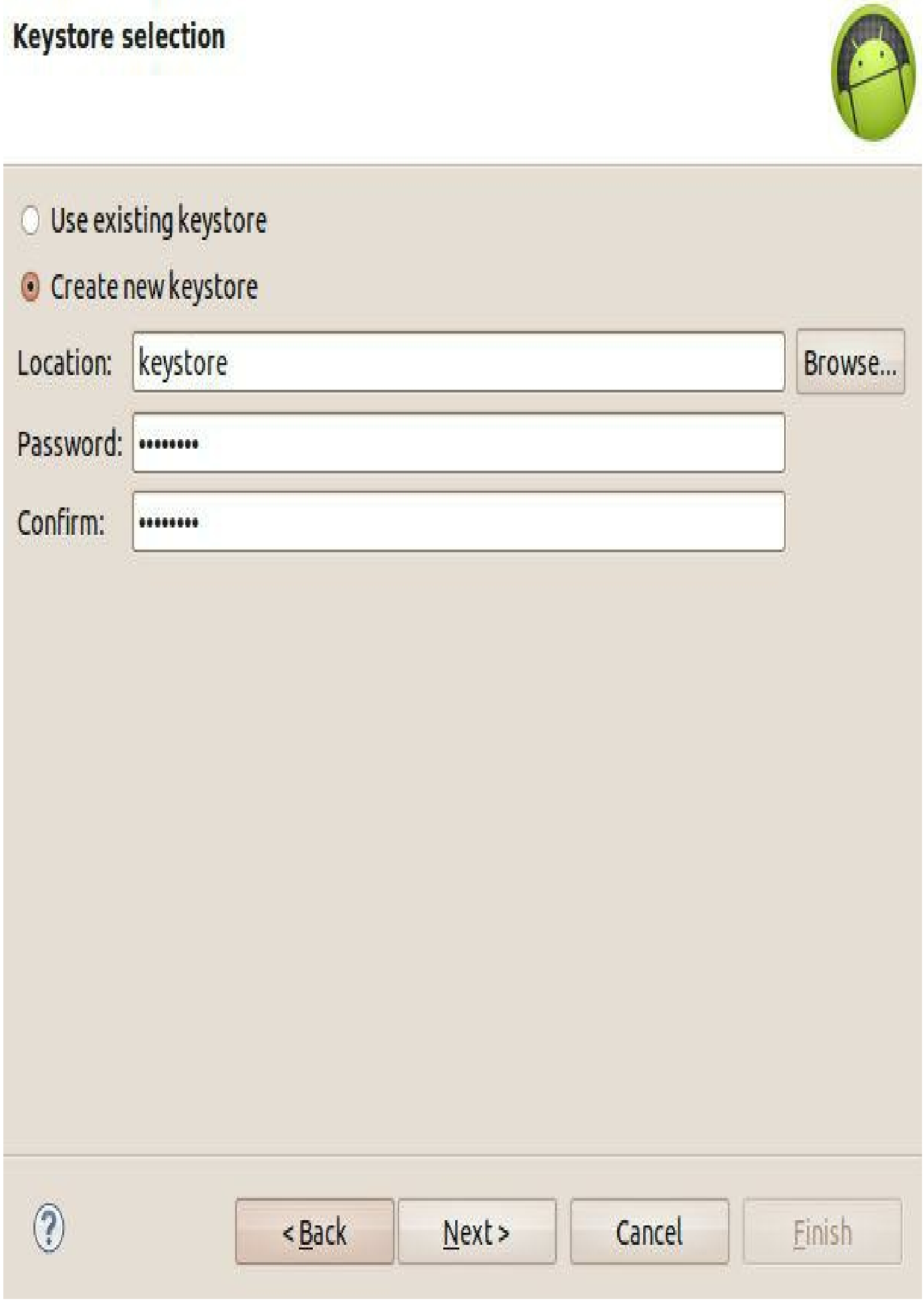
Se la nostra applicazione ha **superato la fase di testing**, allora abbiamo ufficialmente la nostra *release canditate build*. A questo punto dobbiamo generare il package Android, nella pratica un file con estensione *.apk*, e ufficializzarlo con la firma digitale. Per completare facilmente questi passaggi, sia Eclipse che Android Studio offrono appositi *wizard*. Li vediamo nei paragrafi seguenti.

Preparazione della release con Eclipse

Per lanciare il *wizard* su Eclipse è sufficiente cliccare con il tasto destro del mouse sul nostro progetto e selezionare la voce *Export*. Nella finestra che compare selezioniamo l'opzione *Android* e poi *Export Android Application* come mostrato nella seguente figura:

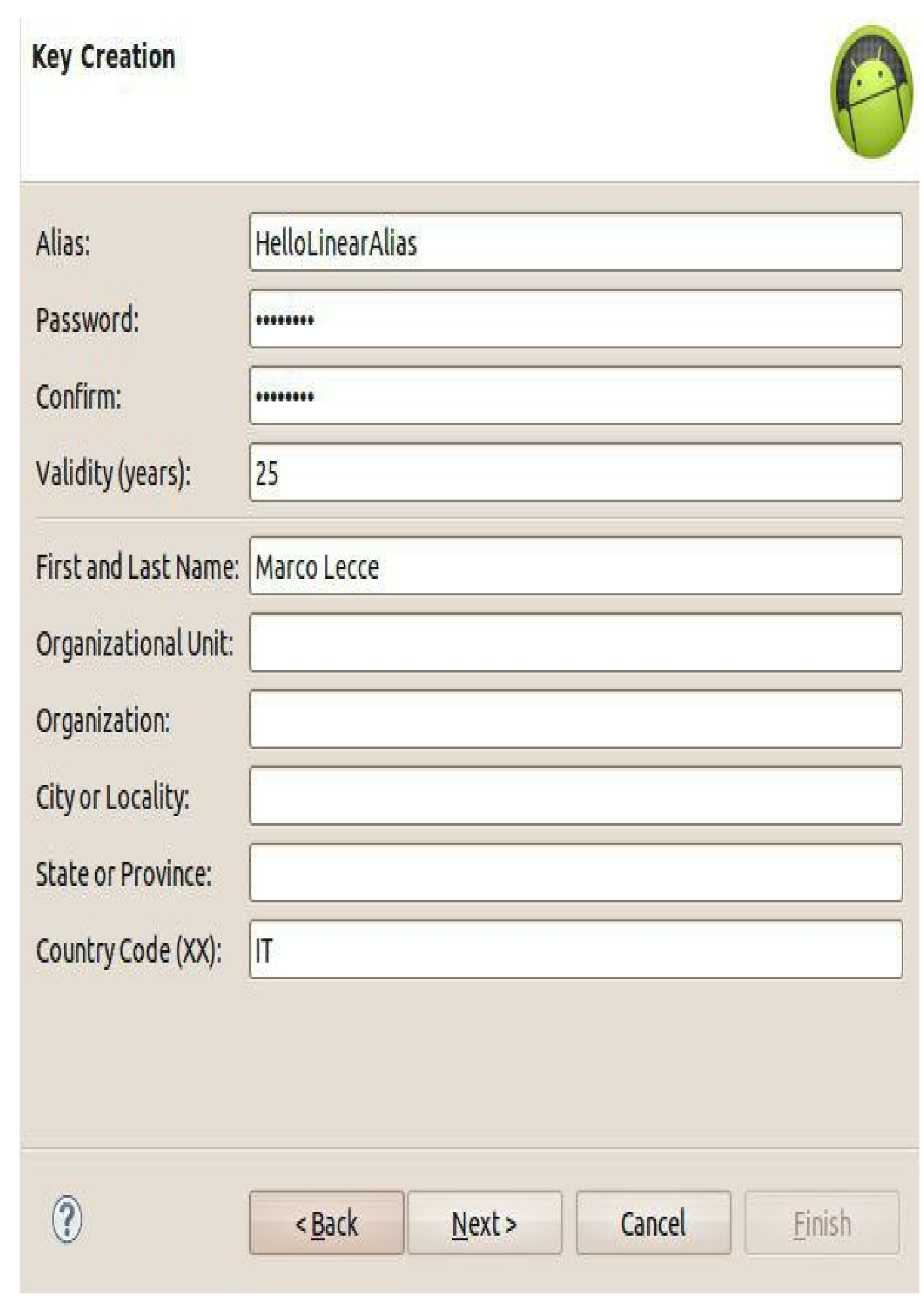

Select
Select an export destination:
type filter text
General
Android
Export Android Application
EJB
Java
Java EE
PHP
Plug-in Development
Remote Systems
Run/Debug
Tasks
< Back
Next >
Cancel
Finish

Clicchiamo su *Next*, controlliamo che il progetto selezionato sia quello effettivamente di nostro interesse (altrimenti ne possiamo selezionare un altro dopo aver cliccato sul bottone *Browse*...), clicchiamo nuovamente su *Next* e arriviamo alla schermata per la selezione della *Keystore*:


Keystore selection
Use existing keystore
Create new keystore
Location:
keystore
Browse...
Password:
Confirm:
< Back
Next >
Cancel
Finish

Scegliamo l'opzione *Create new keystore* (se possediamo già una *keystore* precedentemente creata possiamo anche valutare di utilizzare questa scegliendo l'opzione *Use existing keystore*) e nel campo *Location* inseriamo il percorso del file in cui vogliamo memorizzare la chiave. Inseriamo anche la password per la gestione della chiave, la confermiamo, e clicchiamo su *Next*.

Accediamo dunque alla schermata *Key Creation* in cui dobbiamo inserire alcune informazioni dettagliate sulla chiave, come mostrato dalla seguente figura:


Key Creation
Alias:
HelloLinearAlias
Password:
Confirm:
Validity (years):
25
First and Last Name:
Marco Lecce
Organizational Unit:
Organization:
City or Locality:
State or Province:
Country Code (XX):
IT
< Back
Next >
Cancel
Finish

Il team di Android suggerisce per il campo *Validity* di inserire un valore almeno **uguale a 25**.

Clicchiamo su *Next* per accedere alla schermata *Destination and key/certificate checks*, selezioniamo la destinazione per il file *.apk* ed infine clicchiamo su *Finish*.

Preparare la release con Android Studio

Anche su Android Studio il procedimento di creazione del file *.apk* da distribuire non sarà molto complicato. L'operazione preliminare richiesta, anche in questo caso, sarà la predisposizione di un keystore, idoneo a contenere le chiavi per firmare digitalmente il pacchetto.

Selezioniamo quindi la voce *Generate Signed APK...* dal menu *Build*.

La finestra che si apre, visibile in figura, richiede di indicare una chiave per eseguire la firma del pacchetto.

Sono richieste due password: una per accedere al keystore ed una per la chiave.

Tutto ciò è possibile a patto che si

abbia a disposizione un keystore preesistente. Se così non fosse, sarà necessario crearlo. Per procedere all'inizializzazione di un nuovo "portachiavi", clicchiamo su *Create new*... nella medesima finestra. Nella procedura che porta alla sua creazione verranno richiesti alcuni dati: il percorso nel file system in cui salvarlo, una nuova password per accedervi ed alcuni dati personali da inserire nel relativo certificato.

Una volta in possesso di una chiave si può procedere alla creazione del pacchetto *.apk* firmato digitalmente. Le ultime opzioni da scegliere saranno il percorso di salvataggio del pacchetto ed i l *Build Type*. Per quest'ultimo ci

saranno due opzioni, *release* e *debug*, ma noi sceglieremo la prima.

Note conclusive

Abbiamo così creato un *application package file* pronto per la pubblicazione: prima però di renderlo disponibile all'utente finale attraverso i market, è consigliabile eseguire ancora qualche test sull'installazione del pacchetto *.apk* che abbiamo appena creato.

Proviamo dunque ad installarlo e disinstallarlo sia nell'emulatore sia su uno o più device reali, in modo da verificare che il processo di installazione dell'*.apk* che vogliamo pubblicare sia completo e vada a buon fine.

Conclusi anche questi test sul file *.apk*

possiamo iniziare il procedimento per pubblicare la nostra app nell'Android Market.

Capitolo 58 – Iscrizione a Google e invio dell'app

Oltre ad Android, Google consente anche la pubblicazione delle app sul suo "mercato" ufficiale, dove possiamo trovare software sia gratuiti che a pagamento: **Google Play**.

L'obiettivo di molti programmatori che si avventurano nello studio di una guida come questa è la pubblicazione di una propria app sul market. In questa lezione parleremo di quello ufficiale, fornito da Google; nei capitoli successivi discuteremo delle principali alternative.

Prima di procedere, riflettiamo su

quali possano essere i vantaggi di pubblicare sul market. Probabilmente molti penseranno subito al profitto economico, al poter vendere la propria applicazione. Molti pogrammatori alle prime armi sognano di ideare ed implementare un'applicazione che costi quasi nulla, e risuscire a ricavere milioni dalle vendite. Ma sappiamo bene che ciò è molto più difficile di quanto possa sembrare. Indipendentemente dalla possibilità di realizzare o meno questo "sogno", è bene sottolineare che ci sono altri vantaggi nel possedere (ed utilizzare) un account sviluppatore presso Google Play. Ad esempio, lo si potrebbe popolare di applicazioni gratuite per

dimostrare le proprie abilità, e presentarsi ad un colloquio di lavoro per programmatori con un portfolio ricco e valido. Per non parlare della gratificazione personale derivante o da guadagni per vie diverse dalla vendita, con altre forme di monetizzazione.

Registrarsi

Per iniziare la propria vita di sviluppatore ufficiale Android è necessario effettuare il log-in con un account Google, ad esempio quello usato per Gmail, sull' apposita pagina di accesso.

Fatto ciò, è sufficiente seguire i passi seguenti:

- sottoscrivere il contratto di pubblicazione sull'Android Market. Questo specifica per lo più cosa si può e non si

può fare, le regole da rispettare, i contenuti vietati e così via;

- versare la quota di 25 dollari mediante il sistema di pagamento Google Wallet (operazione che dovrà essere effettuata soltanto una volta).

La pubblicazione

La pubblicazione vera e propria dell'app avverrà dal proprio pannello da sviluppatore dopo aver eseguito il login. Oltre al pacchetto *.apk* firmato, saranno necessari due screenshot dell'applicazione, un'icona dell'app ed una descrizione delle finalità e dei funzionamenti, possibilmente in più lingue. All'occasione della pubblicazione verranno fornite indicazioni circa le dimensioni ed i dettagli di ognuno di questi elementi.

Se si desidera pubblicare **applicazioni a pagamento**, è necessario registrare anche un account commerciale presso **Google Checkout**. Dopo aver

fornito le informazioni richieste, anche questo aspetto sarà completato e si potranno pertanto vendere le proprie creazioni.

Capitolo 59 – Market alternativi

Il market di Google è la piazza principale attraverso cui diffondere le proprie app. Ovviamente, non è detto che un'applicazione Android debba necessariamente essere distribuita mediante un "mercato". Potrebbe essere venduta o scaricata direttamente, da un proprio sito o mediante altri servizi. Indubbiamente, scegliere di distribuirla mediante un canale ufficiale sottopone ad alcuni obblighi e limitazioni, ma d'altro canto offre vantaggi non indifferenti, come la maggiore reperibilità da parte degli utenti, catalogazione più precisa dei contenuti e

strumenti di statistica e analisi.

Oltre all'Android Market di Google esistono mercati alternativi, realtà spesso istituite e gestite dai "grandi" della Rete. Ne vediamo alcuni:

- **Amazon Appstore**: rappresenta un mercato in grande ascesa, forte di tutto il sostegno che può dare un colosso come Amazon, tra i pionieri dell'e-commerce.
  Permette agli sviluppatori di registrarsi gratuitamente e caricare le proprie app da subito,

anche in vendita. Disponibile in circa 200 nazioni, offre diverse caratteristiche interesanti tra cui la possibilità di distribuire applicazioni non native, in HTML5, senza il bisogno di pacchettizzarle, e delle API in grado di integrare strumenti di monetizzazione nell'applicazione. Altro aspetto interessante, testimonianza di quanto valga il supporto di Amazon, è la possibilità di sfruttare la piattaforma

di analisi statistica **Amazon Analytics Service**;

- **SlideME Market** può vantarsi di essere il primo market Android. Nato nel 2008, qualche mese prima di Android Market (oggi Google Play), vuole essere il mercato universale, soprattutto a tutela dei produttori di dispositivi Android che non fanno capo a Google. SAM, il client per questo mercato, è già presente su dispositivi di molti

produttori ed in caso di necessità può essere scaricato dalla home del progetto. Sul sito si presenta un po' come il mercato della libertà: tante nazioni, tanti distributori di dispositivi supportati, zero costi per lo sviluppatore, libertà di monetizzazione. Si può caricare l'app, descriverla a tutto tondo con screenshot, video e testi, oltre a scambiare pareri con gli utenti;

- Samsung ha proposto **SamsungApps** un market

specializzato in applicazioni per i suoi dispositivi, e non limitato al supporto di Android ma anche di Windows, Bada e Symbian. I punti di forza maggiormente pubblicizzati sul sito sono la sicurezza, la qualità delle applicazioni e l'alto grado di compatibilità delle app con i dispositivi Samsung.

Il mondo dei market delle app è in continua espansione, come è lecito aspettarsi per un sistema operativo così

dilagante. Si pensi ad **AppsBrain**, **AppsFire** o **GetJar** per fare qualche altro nome. Ma molti ancora ne nasceranno. Quando lo sviluppatore deve decidere quale sia il market giusto cui affidare la propria app, i fattori che dovrebbe valutare sono tanti, tra cui:

- limitazioni e clausole per l'ammissione dell'app nel market;
- costi da sostenere per lo sviluppatore;
- mezzi di pagamento offerti ai clienti;
- possibilità di

monetizzazione;

- disponibilità di client per l'installazione delle app direttamente su dispositivo;
- diffusione del market.

Capitolo 60 – Modelli di monetizzazione

Avere un account sviluppatore Google permette di caricare un'app e renderla disponibile tramite download dal market. Inoltre, corredando il tutto con un account Google Checkout, è possibile diventare un "mercante" di app, rilasciandole a titolo non gratuito.

Il "come" trarre guadagno da un'app resta una problematica attualmente piuttosto discussa, e venderla non è l'unica via di monetizzazione esistente. Questo capitolo si propone di illustrare per sommi capi le possibilità più comuni per poter far fruttare una propria applicazione, trasformando in moneta

sonante gli sforzi di sviluppo, progettazione e sperimentazione.

Ecco le forme più comuni di "monetizzazione":

- **Vendere un'app**: quando si parla di "vendere" un'app si pensa subito al distribuirla in forma non gratuita. Eppure le statistiche dicono che un modo molto comune e particolarmente proficuo di vendere app è quello di offrirne una versione base gratuitamente,

magari con limitazioni di utilizzo temporali o di funzionalità e proporne versioni più complete in vendita. Ciò non solo risulta una delle principali forme di guadagno, ma permettendo anche all'utente di pagare per un'app che ha suscitato realmente il suo interesse, almeno nella versione gratuita;

- una riedizione nel mondo mobile di un modello tipico dei siti Internet è la **pubblicità**

in-app. Significa che è possibile inserire nell'interfaccia utente della propria applicazione dei banner pubblicitari, occupanti uno spazio ridotto. È possibile includerli nella propria app rivolgendosi a servizi come *AdMob* (uno dei più utilizzati in assoluto) o ad *AdSense* di Google, che si occupano di raggruppare gli inserzionisti in base alle catogorie di interesse e proporre i più adeguati per la propria

app. Affinchè questo meccanismo abbia successo è necessario innanzitutto che l'app sia molto diffusa, quindi gratuita, utile e molto usabile;

altro meccanismo che si può prendere in considerazione è la **vendita di prodotti in-app**. Si tratta di uno strumento molto diffuso e consiste nella possibilità di far acquistare all'utente prodotti o servizi per lo più a carattere digitale, offrendoli durante il normale utilizzo dell'app.

Capitolo 61 – Riepilogo: controlli utente

Questo capitolo offre una panoramica dei più comuni widget che possono essere utilizzati nelle interfacce Android. La finalità non è quella di fornire una documentazione dettagliata, piuttosto una forma di glossario che a colpo d'occhio permetta di individuare un controllo da utilizzare e di averne a disposizione le minime proprietà da configurare per un corretto funzionamento. A tale scopo, i controlli sono stati raggruppati per categorie, e di ognuno verranno illustrati gli attributi XML ed i metodi Java utili nell'uso pratico più comune.

Tra gli attributi XML, non verranno mai citati `layout_width`, `layout_height` e `id`, in quanto già affrontati e di utilizzo comune a tutte le *View* del framework Android.

Il testo

I primi controlli con cui solitamente si fa conoscenza sono i classici “input” per form. Essi permettono all’Activity di svolgere uno dei principali compiti per cui essa è utilizzata: interagire con l’utente e gestire l’immissione dati. Tra questi, la gestione del testo assume una posizione di assoluto rilievo.

TextView

Nella sua forma base rappresenta una label in grado di mostrare stringhe statiche. È il modo più comune per visualizzare contenuti testuali nell'interfaccia utente.

La configurazione avviene in buona parte (ma non unicamente) in XML. Ecco i principali attributi impiegati:

Attributo	Descrizione
`text`	Rappresenta la stringa da mostrare
`textColor`	Rappresenta il colore del testo
`textSize`	Rappresenta la dimensione del testo
`lines`	Rappresenta il numero esatto di righe che deve contenere la TextView
`maxLines`	Rappresenta il numero massimo di righe che può contenere la TextView
`minLines`	Rappresenta il numero minimo di righe che deve contenere la TextView
`textIsSelectable`	Specifica se il testo può essere selezionabile o meno
`text`	Rappresenta
`text`	Rappresenta
`text`	Rappresenta

EditText

È un “discendente” di *TextView* e ne costituisce la sua versione modificabile tramite input dell’utente. Tutte le proprietà viste per la*TextView* sono ugualmente valide anche per *EditText*. Vale la pena sottolineare che di uso molto comune è l’attributo **inputType** che indica il formato dei dati inseriti dall’utente nel campo specifico. Può assumere diversi valori indicanti i tipi di input più comuni, tra cui `text`, `number`, `phone`, `date`, `datetime`, `password`.

AutoCompleteTextView

Viene usato come casella di input (come EditText, da cui eredita gran parte delle proprietà), con la caratteristica che offre una lista di possibili valori per l'inserimento. Questi, mostrati in un menu a tendina man mano che l'utente digita caratteri, vengono selezionati all'interno di una struttura dati presentata da un *Adapter*.

Per usare *AutoCompleteTextView*, è fondamentale:

- **passare un Adapter** opportunamente inizializzato, mediante

codice dinamico Java, tramite il metodo `setAdapter` (è perfetto allo scopo un ArrayAdapter<String>);

- **eventualmente impostare alcuni parametri mediante attributi XML**, come `dropDownHeight` e `dropDownWidth`, che indicano rispettivamente altezza e larghezza del menu a discesa, e `completionThreshold`, un intero che rappresenta il numero minimo di caratteri che l'utente

deve inserire prima che i suggerimenti vengano offerti.

MultiAutoCompleteTextVi

È una classe derivata da *AutoCompleteTextView*. Permette di ricevere suggerimenti riferiti non all'intero testo ma a singole sottostringhe. Oltre ad un *Adapter* impostato mediante `setAdapter`, ha bisogno di un apposito **Tokenizer** che implementi l'interfaccia *MultiAutoCompleteTextView.Tokenizer*.

CheckedTextView

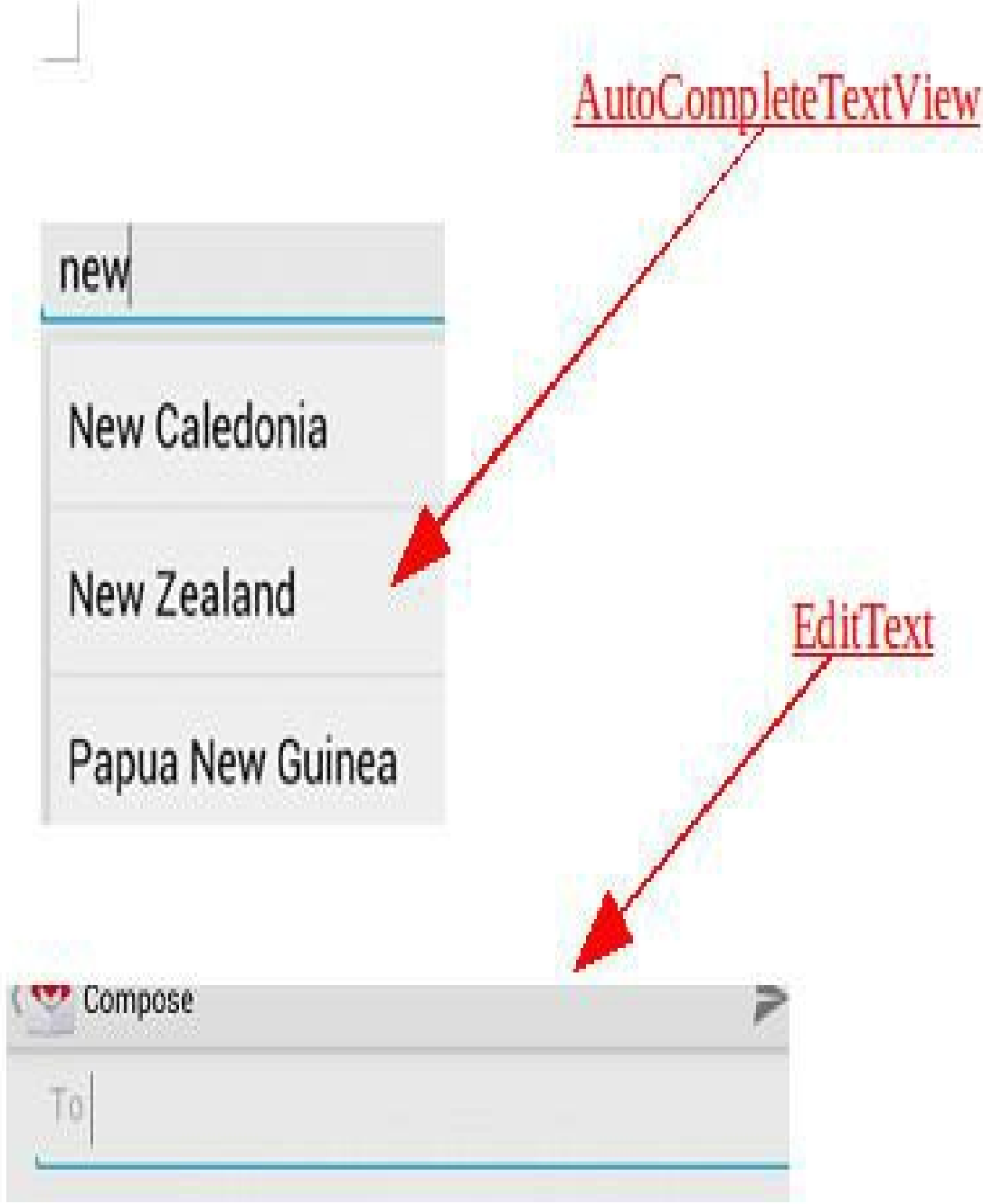
È una *TextView* che implementa l'interfaccia *Checkable*. È utile all'interno delle *ListView* in cui il *ChoiceMode* è stato impostato ad un valore diverso di *CHOICE_MODE_NONE*.

Accorpa in sé le caratteristiche di una *TextView* e di una *CheckBox*. Infatti, con un controllo di questo tipo si possono impostare tutte le caratteristiche del testo (contenuto, stile e proprietà comportamentali) ma si può anche gestire la checkbox inclusa.

Attributi XML molto utili in proposito sono:

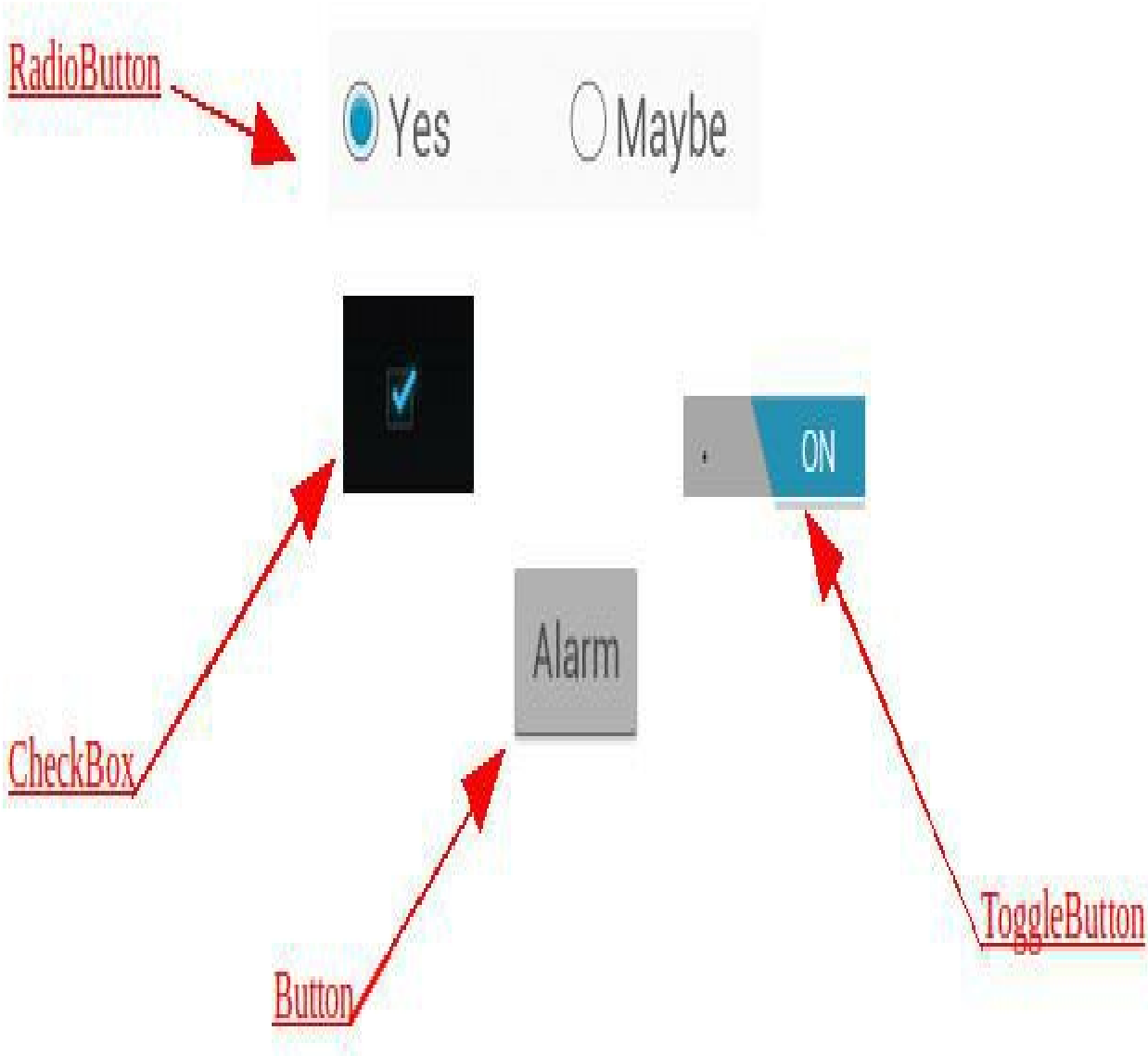
Attributo	Descrizione
`checkMark`	Indica il *Drawable* usato per raffigurare il check su cui cliccare. Per valorizzarlo si possono usare anche stili della piattaforma come ? `android:attr/listChoiceIndicator Multiple`
`checked`	Attributo booleano che indica il valore di inizializzazione della checkbox

L'immagine di seguito riporta un *EditText* ed un *AutoCompleteTextView* utilizzati all'interno di in un layout:


AutoCompleteTextView
new
New Caledonia
New Zealand
Papua New Guinea
EditText
Compose
To

Pulsanti

Sono i controlli tipici di un form, in quanto permettono con un click di attivare un'azione da parte dell'*Activity*, e molto spesso avviano l'elaborazione dei dati inseriti dall'utente.


RadioButton
Yes
Maybe
CheckBox
Alarm
ON
Button
ToggleButton

Button

Il più classico e tipico dei pulsante, al quale può essere associata un'azione in risposta al click. Da questa classe ereditano tutti i tipi di widget che rappresentano pulsanti su Android, e per quanto possa sembrare strano essa eredita dalla classe *TextView*. Per questo motivo, da essa deriva alcune proprietà, come quelle relative al testo.

Gli attributi XML più utili sono i seguenti:

Attributo	Descrizione
`text`	Rappresenta il testo che appare sul pulsante
`onClick`	È una stringa che rappresenta il nome del metodo dell'*Activity* che gestirà il click. Tale metodo deve obbligatoriamente restitire `void` ed accettare in input una *View*, identificabile con il pulsante stesso che ha ricevuto il click

ImageButton

Identico al *Button* nell'impiego e nella configurazione, ma con l'aggiuntiva possibilità di utilizzo di un'immagine al posto del testo. Includendo un *ImageButton* in un layout, piuttosto che l'attributo `text` sarà importante configurarne uno chiamato **src**, che riceve come valore la risorsa *drawable* da raffigurare sul pulsante.

ToggleButton

È un pulsante da utilizzare come interruttore. Tramite questo widget, si può specificare uno di due stati possibili, ed il click esercitato su di esso comporta la transizione da uno stato all'altro.

Al pari di un *Button* supporta l'attributo `onClick` per indicare il metodo dell'*Activity* che gestirà le operazioni corrispondenti alla variazione di stato. Questo controllo permette di indicare due testi alternativi da usare come etichette del pulsante. Si tratta delle proprietà **textOn** e **textOff**. Al cambiamento di stato, l'etichetta passerà da un valore all'altro in

automatico.

RadioButton

Quando l'utente deve scegliere tra alcune opzioni prestabilite, è piuttosto comune presentare nell'interfaccia un gruppo di *RadioButton*. La caratteristica principale dell'intero gruppo è la mutua esclusività tra le opzioni: in altre parole, ne può essere selezionata una sola.

Ogni *RadioButton* supporta le proprietà tipiche di un Button: `text` per impostare l'etichetta, `onClick` per definire il metodo di gestione di un click. È importante che i *RadioButton* vengano inclusi in un **RadioGroup**, un oggetto che maschera un *LinearLayout* arricchendolo dei comportamenti utili alla gestione dei *RadioButton*, tra i

quali:

- `orientation`, che indica (al pari dei *LinearLayout*) se la serie dei controlli contenuti deve svilupparsi in orizzontale o in verticale;

- `checkedButton`, che rappresenta il *RadioButton* impostato a *checked* come opzione di default dell'intero gruppo. È valorizzato con l'id del pulsante prescelto.

CheckBox

È il controllo “booleano” per eccellenza. Il click su di esso generalmente appone o rimuove un segno di “spunta”. L’uso di questo widget consiste nell’impostare ad esso un valore di default, e nel leggerne lo stato.

Nel codice Java questo può essere fatto con i metodi `setChecked(boolean)` e `isChecked()` che, rispettivamente, impostano il valore del controllo e ne leggono lo stato restituendolo in una variabile booleana.

AdapterView e Adapter

Uno degli argomenti centrali delle interfacce utente in Android è la comprensione del **pattern Adapter** e del suo utilizzo mediante particolari *View*, appartenenti alla famiglia delle *AdapterView*. Tutti i controlli di seguito elencati lavorano a contatto con un*Adapter*, pertanto tra i loro metodi più utilizzati in assoluto ci sono quelli relativi a questo aspetto: `setAdapter` per impostare l'*Adapter* da utilizzare, e `getAdapter` per recuperarne il riferimento.

ListView

L a *ListView* rappresenta la tipica visualizzazione di un elenco di valori. Qualora questi siano troppi per la dimensione del controllo, viene attivato in automatico uno scroll verticale.

È possibile dotarla di una **header**, da mostrare in testa ai valori, e di un **footer**, da accodare alla lista. Questi elementi possono essere impostati e rimossi, in Java, mediante i metodi `addHeaderView` e `removeHeaderView` ed i corrispondenti `addFooterView` e `removeFooterView`.

Altro elemento importante da un punto di vista grafico è il **divisore**, un'immagine o un colore che separa gli

elementi della lista in maniera personalizzata. Il divisore può essere impostato sia dinamicamente, mediante il metodo `setDivider(Drawable)`, che in XML con l'attributo `andorid:divider`. Inoltre si può impostare lo spessore del divisore con `setDividerHeight(int)` e `android:dividerHeight`.

Infine, è possibile definire in XML l'attributo `android:entries` che imposta un'array di risorse con cui verrà popolata la lista. Ciò può tornare particolarmente utile nei casi in cui la lista dei valori sia preimpostata.

Infine la ListView permette di definire la modalità di selezione degli elementi – che può essere singola,

multipla e nulla – attraverso la proprietà **choiceMode**, disponibile sia dinamicamente con metodi Java che staticamente come attributo XML.

GridView

Una *GridView* viene utilizzata per rappresentare griglie. Il suo funzionamento di base si avvicina molto a quello della *ListView*. Potremmo sommariamente dire che svolge lo stesso lavoro su "più colonne". Per questo motivo, gli attributi XML di maggiore interesse sono:

Attributo	Descrizione
`numColumns`	Definisce il numero di colonne che devono comporre la griglia
`columnWidth`	Imposta la larghezza delle colonne
`horizontalSpacing`	Definisce la spaziatura di default tra le colonne
`verticalSpacing`	Definisce la spaziatura di default tra le righe

Spinner

È il “menu a tendina” delle interfacce Android. Sul suo funzionamento di base non c’è molto da spiegare in quanto ha molto in comune con la *ListView*. Per utilizzare uno *Spinner* è innanzitutto importante popolarlo. Lo si può fare aggregandovi un *Adapter* esterno o mediante un array di risorse con l’attributo **entries**.

La gestione della selezione di un elemento può essere svolta impostando un apposito listener con il metodo `setOnItemSelectedListener`, implementando altresì l’override del metodo `onItemSelected` per specificare il codice da eseguire.

Informazioni sulla posizione o il valore dell'elemento selezionato possono essere recuperate, in Java, con i metodi `getSelectedItem()`, `getSelectedItemId()` e `getSelectedItemPosition()`.

ExpandableListView

È un *AdapterView* che mostra un elenco di risultati suddivisi in gruppi. Tutta la gestione si basa sulla possibilità di creare un elenco di gruppi, facendo sì che, cliccando su una singola voce, si apra una lista di elementi attinenti al gruppo scelto.

Risulta un po' più complicata la gestione dell'*Adapter*, soprattutto se si decide di crearne uno personalizzato che estende *BaseExpandableListView*. In questo caso, i metodi `getGroup`, `getGroupCount`, `getGroupId`, `getGroupView` serviranno a gestire i gruppi, mentre i metodi `getChild`, `getChildrenCount`, `getChildId` e

`getChildView` svolgeranno le corrispondenti operazioni relative agli elenchi di elementi attinenti ai gruppi.

Misurare il tempo

Il framework di Android include anche diversi **controlli per la gestione delle date e degli orari**, sia come input utente che semplicemente per la visualizzazione.

DatePicker e TimePicker

I picker sono le tipologie di controlli più comunemente utilizzati nelle interfacce utente per impostare data e ora. Il loro scopo è quello di permettere l'inserimento di questi dati da parte dell'utente in un formato corretto anche in relazione al proprio fuso orario. Ciò è basilare affinché i dati vengano interpretati correttamente.


Set time
7
29
8
:
30
AM
9
31
PM
Cancel
Set


Set date
Sep
06
2010
Oct
07
2011
Nov
08
2012
Cancel
Set

AnalogClock e DigitalClock

Sono degli orologi veri e propri. Mostrano l'ora attuale del sistema, il primo in forma analogica –con le lancette – il secondo in forma digitale, quindi riportando come testo l'informazione (ad esempio "8:23:15 PM").

Per utilizzarli è sufficiente collocarli nel layout e, totalmente in autonomia, continueranno a visualizzare l'ora aggiornata del sistema.

È importante sottolineare che non possono essere usati per inserire informazioni data/ora; per questo scopo, come già detto, esistono appositamente i

picker.

Chronometer

È un cronometro che permette di misurare il trascorrere del tempo. Una volta collocato nel layout, va gestito mediante codice Java, specialmente con i due metodi start e stop che, rispettivamente, avviano e fermano il cronometro.

Altri aspetti importanti del cronometro sono il tempo di partenza, che può essere gestita con i metodi **setBase** e **getBase**, ed il formato in cui l'informazione temporale viene espressa, con **setFormat** e **getFormat**. Quest'ultimo aspetto può essere definito staticamente in XML mediante l'attributo `android:format`.

Le “barre”

Le varie barre che esistono in Android ricalcano gli analoghi indicatori che siamo abituati ad utilizzare o vedere all’opera nel web e nelle applicazioni desktop. Lo scopo è il medesimo e le varie tipologie, qui di seguito indicate, ne mostrano le diverse incarnazioni.

ProgressBar

La *ProgressBar* è la tipica barra che indica l'avanzamento di un'operazione in corso di svolgimento. Assume solitamente due forme: lo "spinner" (da non confondere con il controllo *AdapterView*) rappresentato da una forma circolare che gira, e la barra orizzontale vera e propria. La differenza tra le due visualizzazioni non è solo grafica, ma consente di adattarsi alla possibilità di quantificare il task in esecuzione: lo spinner è più adatto a lavori indeterminati (download di file dalla rete, per esempio), mentre la barra si può usare per qualunque lavoro in cui il tempo o la mole di lavoro residuo può essere quantificata.

Gli attributi XML più comuni nella definizione della *ProgressBar* sono:

Attributo	Descrizione
`progress`	Indica il livello di completamento cui si è arrivati, adatto alla forma "a barra"
`max`	Indica il massimo valore della barra, indica il completamento del task
`indeterminate`	Valore booleano che stabilisce se la barra deve includere l'indicazione del livello di completamento
`style`	Indica la forma che deve avere la barra. Non definendo lo stile, la *ProgressBar* sarà uno spinner; indicando uno stile (ad esempio uno stile orizzontale, specificando il valore `@android:style/Widget.ProgressBar.Horizontal`) sarà mostrata una barra vera e propria

Per indicare le precedenti proprietà dinamicamente in Java, esistono opportuni membri di classe e metodi. In particolare, molto comune è utilizzare il metodo **setProgress**, mediante il quale sarà possibile aggiornare il valore attuale della barra, ricalcando l'avanzamento del task in corso.

SeekBar

È una classe derivata dalla *ProgressBar*, con la differenza che ne è una versione modificabile. In questo widget, infatti, l'utente può cambiare il livello raggiunto trascinando un'immagine che funge da indicatore. Un esempio tipico è il lettore audio, in cui si potrebbe spostare indietro l'indicatore della *SeekBar* per riascoltare una porzione di audio precedente.

Tutti le sue caratteristiche principali sono quelle della *ProgressBar*, ma per il suo scopo specifico vale la pena tenere a mente:

- l'attributo **thumb**, che può essere valorizzato con l'ID di una risorsa *drawable*, permettendo di scegliere la forma dell'indicatore trascinabile;
- il listener **OnSeekBarChangeListe** che dal codice Java consente di associare ed eseguire un'azione tutte le volte che l'utente modifica la posizione dell'indicatore.

RatingBar

La classe *RatingBar* deriva da *SeekBar*, quindi indirettamente anche da *ProgressBar*. Permette di visualizzare una barra di progresso modificabile, che non è però di forma orizzontale, bensì è costituita da una serie di simboli, tipicamente stelle. Il funzionamento è simile a quello della *SeekBar*, ed il suo campo di applicazione è tipicamente correlato alla possibilità di assegnare un voto discreto (tipicamente, come già detto, un numero di stelle).

L'attributo **numStars** offre la possibilità di scegliere da "quante stelle" deve essere costituita la *RatingBar*.

Immagini e Web

ImageView

È il widget che contiene un'immagine. Il suo attributo principale è **src**, che indica la sorgente dell'immagine da visualizzare (tipicamente una delle risorse dell'applicazione). Il metodo Java che serve ad impostare il *Drawable* da rappresentare è `setImageResource(int)`, e l'intero richiesto come parametro è l'ID della risorsa.

WebView

È un browser vero e proprio inserito all'interno del layout. Nella guida gli è stato dedicato un intero capitolo; quindi, in questa sede, ne riassumiamo le caratteristiche principali. Oltre all'opportuno dimensionamento del controllo, l'operazione principale da svolgere è l'assegnazione dell'URL di cui mostrare il contenuto online. Lo si può fare con il metodo Java **loadUrl**, al quale verrà passata una stringa rappresentante un indirizzo web, o con **loadData** al quale, tra l'altro, si deve passare una stringa contenente codice HTML da visualizzare.

È importante ricordare che per

consentire l'accesso alla rete da parte dell'applicazione è necessario **dichiarare nel manifest** l'apposita permission, cioè `android.permission.INTERNET`.

Capitolo 62 – L'Application Context

L'*application context* è un **elemento centrale e importante per tutte le funzionalità** disponibili al più alto livello applicativo: possiamo utilizzare l'*application context* per accedere alle risorse e alle impostazioni condivise tra istanze di diverse *Activity*.

Una volta recuperato l'*application context*, ad esempio con il seguente codice:

```
Context context = getApplicationContext();
```

possiamo utilizzarlo per **accedere ad un ampio insieme di caratteristiche e**

servizi a disposizione dell'applicazione. Ad esempio possiamo richiamare il metodo `getSharedPreferences()` dell'*application context* per recuperare le preferenze condivide dell'applicazione, oppure possiamo richiamare il metodo `getResources()` per recuperare le risorse dell'applicazione:

```
String hello = context.getResources().getString
```

Visto che la classe `Activity` deriva dalla classe `Context` possiamo utilizzare il riferimento `this` oppure richiamare esplicitamente l'*application context*.

Potremmo dunque riscrivere l'esempio precedente come segue:

```
String hello = getResources().getString(R.string
```

L'*application context* lo utilizziamo anche per:

1. Lanciare istanze di *Activity*.
2. Gestire le directory, i database e i file protetti dell'applicazione.
3. Richiamare i gestori di servizi di sistema (e.g. servizio di localizzazione).

4. Applicare e controllare i permessi a livello applicativo.
5. Richiamare gli asset dell'applicazione.

Oltre all'*application context* **abbiamo a disposizione altre due opzioni per richiamare e lanciare istanze di** *Activity*: implementare il lancio di una *Activity* nel file Manifest, oppure lanciare un'*Activity* figlia, da un'altra *Activity* padre per riceverne un risultato.

Capitolo 63 – Esecuzione e debug su Eclipse ADT

Prima di lanciare un'app Android tramite Eclipse, può essere necessario approfondire il processo di creazione di una configurazione di "Run" o di "Debug" per il progetto. Sebbene le ultime versioni di ADT semplifichino di molto questo precesso, i dettagli della creazione possono essere utili in alcuni casi specifici. Vediamo quindi i passi da completare per creare una configurazione di "Debug", in genere più adatta alla fase di sviluppo:

1. Su Eclipse, nel menu

principale selezioniamo il percorso *Run / Debug Configurations*;

2. Nell'elenco disponibile sulla parte sinistra della finestra di dialog che si apre eseguiamo un doppio click sulla voce "Android Application" per creare una nuova configurazione;

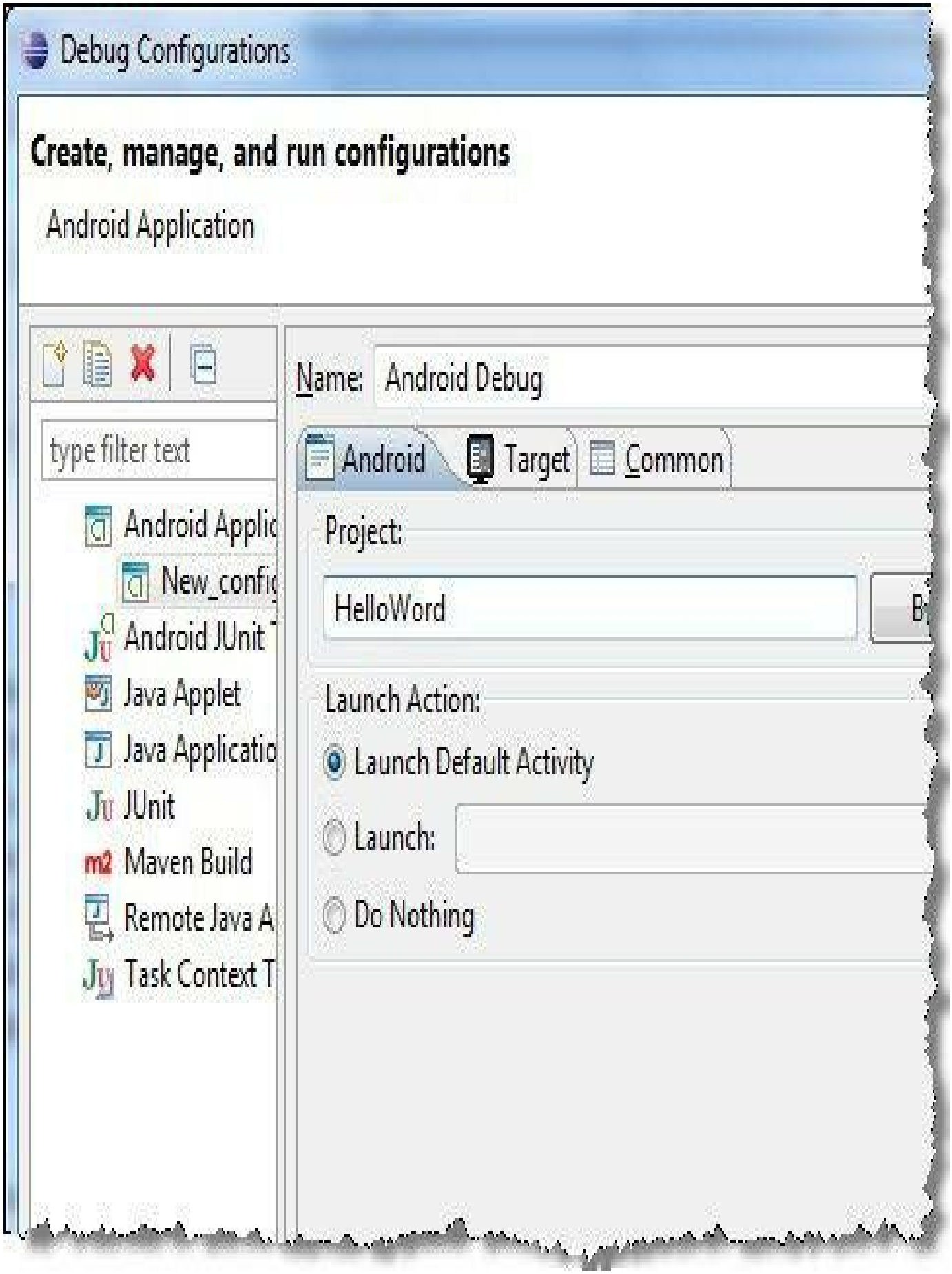
3. Cambiamo il nome della configurazione appena creata da "New_configuration" ad "Android Debug" (o altro nome);

4. Clicchiamo sul pulsante *Browse* e scegliamo il progetto che vogliamo debuggare, nel nostro caso “HelloWord”;

5. Spostiamoci sulla scheda *Target* e controlliamo il target associato al nostro progetto. Se lasciamo la spunta su *Automatic* verrà scelto in automatico il target adatto al progetto lanciato, altrimenti possiamo scegliere di spuntare la voce *Manual*:

scegliendo questa opzione ogni volta che lanciamo un progetto con la configurazione "Android Debug" ci verrà chiesto di selezionare un target. Questa opzione è molto utile e comoda quando dobbiamo testare la nostra app su differenti smartphone o emulatori;

6. Clicchiamo sul pulsante *Apply* per confermare le modifiche.


Debug Configurations
Create, manage, and run configurations
Android Application
type filter text
Android Applic
New_confic
Android JUnit
Java Applet
Java Applicatio
JUnit
Maven Build
Remote Java A
Task Context T
Name: Android Debug
Android
Target
Common
Project:
HelloWord
Launch Action:
Launch Default Activity
Launch:
Do Nothing

La **creazione di una configurazione per il running delle app** è un processo molto simile a quello che abbiamo appena visto per la creazione della configurazione di debugging. Da un punto di vista pratico ciò che cambia è al punto 1 della precedente lista, dove al posto della voce *Run / Debug configurations* dovremmo seguire il percorso *Run / Run Configurations*.

Da un **punto di vista funzionale** invece ciò che cambia principalmente nell'utilizzare una configurazione di running piuttosto che una di debugging è che se usiamo quella di debugging il debugger di Eclipse sarà collegato alla nostra applicazione, facilitando il debug del nostro codice anche grazie

all'utilizzo dei breakpoint.

Utilizzando il plugin ADT di Eclipse, quando eseguiamo il running o il debugging della nostra applicazione succede quanto segue:

1. Viene compilato il progetto corrente e convertito in un eseguibile Android (.dex);
2. Gli eseguibili e le risorse esterne della applicazione vengono organizzate in un package Android (.apk);

3. Il device virtuale selezionato viene fatto partire (se non è già stato fatto partire prima, altrimenti questo punto non viene eseguito);
4. L'applicazione viene installata nel device vituale selezionato;
5. L'applicazione viene fatta partire nel device vituale.

Capitolo 64 – Eseguire test sull'emulatore e sullo smartphone

In questa lezione vedremo come eseguire il debug di un’app usando, prima, l’emulatore di Eclipse e, dopo, un dispositivo fisico reale (ad esempio uno smartphone).

Test nell'emulatore

Per lanciare e testare la nostra app nell'emulatore, tutto quello che dobbiamo fare è cliccare sul pulsante *Debug* che troviamo nella toolbar di Eclipse; oppure premere il tasto *F11* o scegliere la voce *Debug* dal menu *Run*.

Una volta cliccato sul pulsante *Debug* verrà lanciato l'emulatore: al primo lancio è necessario aspettare un po' di tempo prima di poter interagire con il device virtuale, dunque aspettiamo con calma che lo startup si concluda (anche alcuni minuti su computer meno potenti).

Se abbiamo fatto tutto correttamente, nel nostro emulatore vedremo la nostra app in esecuzione. La figura seguente

mostra un esempio molto semplice, in cui un “Hello World!” è stato eseguito sull’emulatore:

Test nello smartphone

Dopo aver testato la nostra applicazione nel device virtuale, vediamo come essa si comporta su un device reale.

Per procedere abbiamo bisogno di un cavo USB per collegare il nostro smartphone al PC e completare la seguente procedura:

1. In Eclipse selezioniamo *Run / Debug Configurations*;
2. Clicchiamo due volte su *Android Debug*;
3. Clicchiamo sulla

scheda *Target* e impostiamo *Deployment Target Selection Mode* su *Manuale*“. Selezionare la voce *Manual* che ci permetterà di scegliere se eseguire il debug nell’emulatore o nel device;

4. Clicchiamo sul pulsante *Apply*;
5. Colleghiamo attraverso una porta e un cavo USB il nostro smartphone al PC.
6. Nel caso in cui

stessimo lavorando in ambiente Linux (es. Ubuntu) è necessario eseguire qualche passaggio ulteriore:

- Acce alla directory *platform-tools*: la troviamo nella directory in cui abbiamo installato

l’SDK di Android;

- Esegu il “kill” del server “adb” lanciando il comando *sudo ./adb kill-server*;
- Riavv il server “adb” con il comando *sudo*

./adb start-server. In questo modo il server avrà i permessi necessari per accedere al device collegato al Pc;

1. Facciamo clic sul pulsante *Debug* per

concludere la configurazione.

A questo punto comparirà la finestra *Android Device Chooser*, in cui vengono visualizzati i device, virtuali e/o reali, di cui possiamo disporre nel nostro PC e che possiamo scegliere per eseguire il debug dell'applicazione. Il nostro device deve essere abilitato per il debug attraverso la connessione USB. Per attivare questa configurazione, selezioniamo il menu *Impostazioni* dello smartphone, poi la voce *Applicazioni* e qui impostiamo la seguente configurazione:

1. Attiviamo l'opzione *Origini sconosciute*;
2. Selezioniamo la voce *Sviluppo*;
3. Nelle opzioni disponibili in questa sezione attiviamo le voci *Debug USB* e *Rimani attivo*: quest'ultima opzione disabilita il controllo sull'attivazione dello schermo, lasciandolo sempre attivo.

Per concludere il processo è sufficiente fare doppio clic sul device in cui si vuole eseguire l'applicativo nella

finestra *Android Device Chooser*.

Eseguito l'ultimo passaggio, Eclipse **installerà l'applicazione Android nel dispositivo** che abbiamo selezionato dall'elenco dei device disponibili (reali o virtuali) e la eseguirà. Se abbiamo eseguito questa procedura per la stessa applicazione "Hello World!" vista nell'esempio dell'emulatore, nel nostro smartphone comparirà una schermata molto simile a quella rappresentata nella figura precedente.

Rimane da sottolineare ancora una questione: ogni volta che finiamo di lavorare sullo smartphone dobbiamo ricordarci di disabilitare le impostazioni attivate sopra. L'attivazione permanente

dello schermo, se non disabilitata, ovviamente porta ad un rapido scaricamento della batteria, mentre l'attivazione delle altre opzioni al di fuori del contesto di sviluppo lascia aperte notevoli falle per la sicurezza del nostro device.