Impariamo a sviluppare su piattaforma Android applicazioni per smartphone, tablet e dispositivi embedded che ci permetteranno di interfacciarci col mondo elettronico esterno e con i nostri progetti. Prima puntata.

di Andrea Chiappori

1

PROGRAMMIAMO CON ANDROID

Android nasce nel 2007 da Google come un sistema operativo open-source per dispositivi embedded e negli anni successivi ha ottenuto sempre enormi consensi, tali da renderlo attualmente il più valido concorrente di iOS di Apple.
Ad oggi, sono moltissimi gli smartphone ed i tablet sui quali è installato Android, ed ultimamente stanno nascendo anche molti sistemi di sviluppo in grado di supportare questo sistema operativo che, essendo open-source, permette un facile adattamento ed un'ampia modularità verso vari dispositivi hardware, diversi tra loro per funzionalità e caratteristiche, pronti a soddisfare ogni tipo di esigenza.

Breve introduzione al corso

In questo corso a puntate, ci addentreremo nel mondo Android e prenderemo familiarità con gli strumenti forniti gratuitamente dalla comunità di sviluppo di questo nuovo sistema operativo per poi dare vita alle nostre prime applicazioni su qualsiasi smartphone o tablet e con le quali saremo in grado di controllare remotamente i nostri progetti elettronici, impartire comandi e ricevere da essi utili informazioni che potranno essere visualizzate sul display del nostro dispositivo.
La comunità di sviluppo Android è, infatti, sempre in continua evoluzione (come è facilmente riscontrabile visitando il sito *http://developer.android.com*) e ci fornisce tutti gli strumenti e le librerie di codice per prendere il pieno possesso del nostro smartphone e di tutti i suoi sensori e moduli interni. Facilmente il dispositivo, smartphone o tablet, su cui svilupperemo la nostra applicazione, avrà un modulo bluetooth inter-

KIT DI SVILUPPO PER DISPOSITIVI ANDROID

Android non è solo un progetto software, ma coinvolge anche l'hardware e lo dimostra una serie di piattaforme di sviluppo nate attorno a questo sistema operativo open source per sfruttare e controllare svariate periferiche. Un esempio ci viene fornito dalla scheda di sviluppo *IOIO Android* (**Fig. 1**) che presenta una porta USB funzionante come host e a cui andrà collegato il cavetto dati dello smartphone; da Google Play (il market Android)

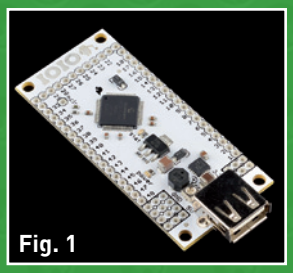


Fig. 1

sarà possibile scaricare applicazioni sul proprio dispositivo per prenderne il pieno controllo o realizzarne di proprie.
Anche Microchip si è affacciata sul mondo Android presentando un kit di sviluppo di questo tipo basato sul microcontrollore a 16 bit PIC24F, come

Fig. 2

visibile in **Fig. 2**.
Non poteva certo mancare una delle più recenti board della famiglia Arduino che oltre all'interfaccia USB Host per interfacciarsi con Android presenta, rispetto alle schede precedenti, un più potente ATmega2560 ed una connetività maggiore verso l'esterno.
Si chiama Mega ADK For Android ed è compatibile con i numerosi shield per Arduino (**Fig. 3**).

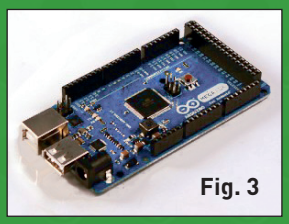


Fig. 3

no, così come un GPS integrato, il Wi-Fi, un accelerometro a tre assi, una fotocamera, oltre, ovviamente, al display e ad un pannello touch, senza dimenticare l'altoparlante, un microfono e il supporto per una memoria esterna micro SD di qualche GByte. Tutte caratteristiche che forse oggi possono apparire scontate ai più, ma che non troppo tempo fa, si sarebbero potute trovare insieme solo in una scheda di sviluppo professionale di fascia alta. Con Android ed una base di programmazione avremo la possibilità di mettere insieme e pilotare questi moduli a nostro piacimento per realizzare interfacce verso il mondo esterno e conferendo valore aggiunto ai nostri progetti elettronici. In particolare realizzeremo, nelle puntate successive, una scheda dotata di modulo bluetooth e di alcuni relé che piloteremo dal nostro smartphone Android ed una con a bordo un modulo Wi-Fi per permettere scambio di dati tra due o più dispositivi.
Esistono poi, anche schede di sviluppo realizzate da terze parti su cui troviamo microcontrollori Microchip o Atmel che si interfacciano direttamente tramite cavetto USB a qualsiasi dispositivo che potremo personalizzare a nostro piacimento. Alcune di queste demoboard sono presentate nel Box "Kit di sviluppo per dispositivi Android".

Programmare su Android

Per programmare con Android viene utilizzato il linguaggio java. In realtà occorre fare prima una breve precisazione perchè il linguaggio usato non è esattamente lo stesso java tradizionale che siamo abituati a conoscere. Normalmente dopo la compilazione di un programma scritto in java viene generato un byte code che viene eseguito da una JVM (Java Virtual Machine). Possiamo pensare la JVM come una implementazione software che permette al nostro codice java di colloquiare col sistema operativo.
Su Android il byte-code generato dopo la compilazione non è proprio lo stesso di quello generato da un compilatore java ed il 'motore' a cui questo viene passato non è una JVM classica, ma una DVM ovvero Dalvik Java Machine; ogni dispositivo Android ha una propria DVM in grado di eseguire questo particolare byte-code. Senza scendere troppo nei dettagli, possiamo affermare che la differenza tra le due macchine virtuali consiste nel fatto che la DVM è pensata proprio per dispositivi embedded in cui la quantità di memoria non potrà essere così elevata come quella di un PC e quindi è ottimizzata per migliorarne le prestazioni.
Di sicuro, a livello di programmazione, chi già conosce java non avrà problemi ad entrare nell'ottica Android , chi invece preferisse comunque programmare in C o C++ potrà essere soddisfatto ugualmente perchè è disponibile un tool di sviluppo (sempre tramite developer.android.com) chiamato NDK

(Native Development Tool) che permetterà di scrivere le nostre applicazioni in codice nativo aumentando così prestazioni, velocità e facilitando anche la riusabilità del proprio codice C o C++.

Platform SDK Android

Ora possiamo incominciare ad installare sul nostro computer (PC, Mac o macchina Linux) tutto il necessario per sviluppare su piattaforma Android.
I passi da seguire sono molto semplici, prima di tutto occorrerà installare (per chi non l'avesse già fatto) il run-time java (JDK) essenziale per la compilazione dei progetti.
Successivamente dovremo installare la toolchain di sviluppo per Android, chiamata Platform SDK, dal seguente sito *http://developer.android.com/sdk*.
Durante l'installazione potremo scegliere quali pacchetti installare tra quelli disponibili in rete. Esistono numerose versioni di Platform SDK corrispondenti alle versioni di Android disponibili sul mercato; è possibile anche sceglierle tutte, ma almeno agli inizi e per non creare troppa confusione consigliamo di concentrarsi solo su una, scegliendola in base alla versione presente nel sistema (smartphone o tablet) a nostra disposizione e sul quale svilupperemo in futuro.
Per avere un' idea generale sulle versioni di Android e dei loro nomi in codice, rimandiamo al Box "Android, versioni e nomi in codice", per adesso basti sapere che le versioni 1.5 e 1.6 sono ormai obsolete, le versioni dalla 2.1 alla 2.3 nascono principalmente per dispositivi smartphone mentre le versioni 3.0, 3.1 e 3.2 sono dedicate ai tablet. Infine le più nuove versioni 4.0 possono funzionare indipendentemente sia su tablet sia su smartphone pur scrivendo gli stessi 'frammenti' di codice, oltre ad avere funzionalità in più come ad esempio il supporto per il *Wi-Fi Connect* ovvero la possibilità di connettere direttamente due dispositivi Wi-Fi senza ricorrere ad un router centrale.
Questo non vuol dire, però, che un' applicazione sviluppata usando le librerie 2.1 non possa "girare" su uno smarthpone su cui è installato un Android 2.3, ma anzi potrà funzionare correttamente anche su un tablet, ovviamente non sfrutterà le funzionalità in

ANDROID, VERSIONI E NOMI IN CODICI

Fig. 4

Ad ogni versione del sistema operativo Android, rilasciata da Google, viene associato un nome in codice sulla base di prodotti dolciari. In contemporanea, al *Googleplex*, il quartier generale di Google (in California), viene realizzata una scultura che lo rappresenta (**Fig. 4**). Come si può notare dalla **Tabella 1**, le iniziali del nome di ogni versione sono in ordine alfabetico.
La versione 2.1 (Eclair) ha mantenuto lo stesso nome per via di alcuni problemi della 2.0 che rendevano il sistema instabile.
Le versioni HoneyComb sono state pensate appositamente per i tablet, mentre le versioni IceCream, uscite di recente solo su alcuni dispositivi di fascia alta, hanno un' interfaccia unificata sia per smartphone sia per tablet.
Il più grande salto di versione in termini di miglioramento di prestazioni e migliorie è stato nel passaggio da Eclair a Froyo, in cui abbiamo possibilità di trasformare il dispositivo Android (connesso in rete) in un hotspot Wi-Fi (questa proprietà è chiamata *tethering*) ed il trasferimento tramite modulo bluetooth dei dati e non solo della voce come nelle versioni precedenti. Quest'ultima caratteristica è utilissima per i nostri scopi qualora volessimo usare il nostro dispositivo Android per controllare a distanza i nostri progetti, come vedremo in una delle prossime puntate.
È comunque possibile aggiornare il proprio sistema Android ad una versione successiva rispetto a quella preinstallata, consultando il sito del produttore del vostro dispositivo.
Essendo Android open-source, i più smanettoni potranno anche modificare ed installare altri sistemi operativi alternativi, ma prendendosi i rischi di avere un dispositivo instabile.

Tabella 1

VERSIONI ANDROID		
Numero versione	Nome in codice	API Level
1.0	Apple Pie	1
1.1	Banana Bread -	2
1.5	Cupcake -	3
1.6	Donut	4
2.0.x	Eclair	5, 6
2.1.x	Eclair	7
2.2.x	Fro Yo (Frozen Yogurt)	8
2.3.x	Gingerbread	9, 10
3.x	Honeycomb	11, 12, 13
4.x	Ice_Cream_Sandwich	14, 15

COMPILARE DA LINEA DI COMANDO

È possibile compilare un progetto direttamente da linea di comando grazie a due semplici script.
Per prima cosa occorre creare il progetto con tutte le sue directory necessarie lanciando il seguente comando di esempio:

Android.bat create project –target 1 –name AndroidApplication *–path ./my*Project *–activity* AndroidActivity *--package* com.packagename.android.

Di seguito una breve descrizione dei parametri inseriti:

–target	specifica la versione delle librerie usate; ad esempio per le librerie API 8 (Android 2.2) il valore deve essere 1.
–name	indica il nome del progetto
–path	indica il percorso della directory in cui verrà creato il progetto con le sue sotto-directory.
–activity	specifica il nome della classe principale che verrà creata in automatico
–package	deve essere un nome univoco nel caso si voglia pubblicare l'applicazione sul market Google Play e normalmente per questo motivo si usa un indirizzo web al contrario, ad esempio com.packagename.android

A questo punto non ci resta che lanciare il secondo comando:

ant debug o *ant release*

e seguire a video i passi della compilazione. Al termine di essa, nella sotto cartella *bin* troveremo il risultato finale, ovvero un file con estensione .apk e vedremo in seguito come poterlo testare direttamente sul dispositivo finale o sul proprio pc, tramite emulatore.
Per comodità consigliamo di aggiungere alla variabile di ambiente *PATH* di windows la directory in cui abbiamo installato i tool di Android in modo da poter eseguire i comandi da ogni posizione.

più offerte dalle versioni successive. Per questo motivo e per il fatto che attualmente nella grande maggioranza degli smartphone sono installate le versioni 2.2 o 2.3, consigliamo di scegliere come libreria di sviluppo SDK la 2.2, ed eventualmente installare successivamente versioni più recenti.
Tra i vari pacchetti installabili sono presenti anche alcune librerie di Google che possono essere usate se si desidera impiegare nelle proprie applicazioni la funzionalità di navigazione nelle mappe geografiche sfruttando il motore Google Maps. Al termine dell' installazione avremo a disposizione i tool necessari per compilare la prima nostra applicazione, anche se per ora solo da linea di comando. Infatti, grazie ad una serie di comandi presenti nella sotto directory *tools* che ora troveremo nel percorso in cui abbiamo precedentemente scelto di installare SDK-Android, possiamo creare un progetto minimale e successivamente compilarlo. Maggiori dettagli al riguardo li potete trovare nel Box "Compilare da linea di comando".

ECLIPSE, l'ambiente di sviluppo grafico

Compilare da riga di commando è utile per capire i vari passaggi ed i file che entrano in gioco durante la compilazione, ma se volessimo avere a disposizione un pratico ambiente di sviluppo integrato che ci permetta di editare i sorgenti, compilare e debuggare su un emulatore o direttamente sul dispositivo fisico in tempo reale, controllando il flusso del programma, possiamo ricorrere ad Eclipse, l'IDE di sviluppo più usato per programmare dispositivi **embedded** e creato in Java. Esistono diverse versioni di Eclipse scaricabili gratuitamente a questo indirizzo *http://www.eclipse.org/downloads*,

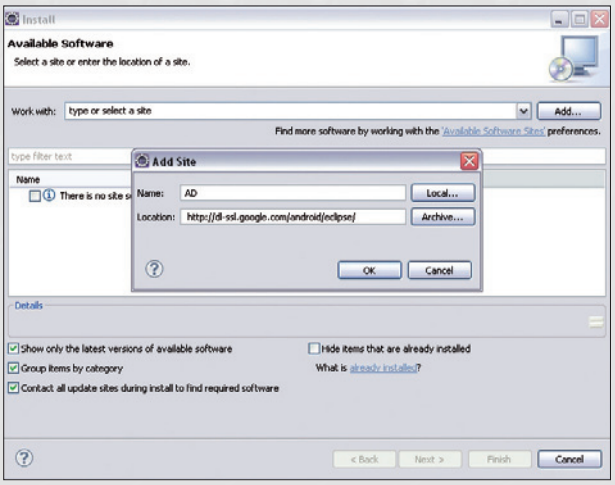


Fig. 5

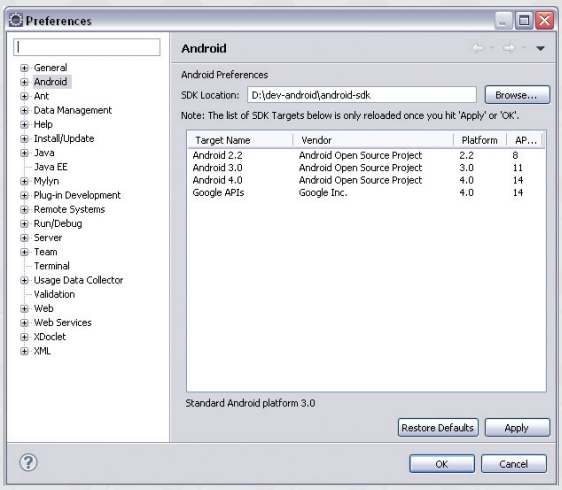


Fig. 6

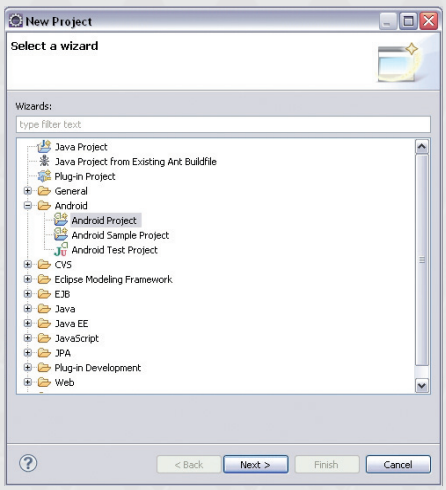


Fig. 7

ma per i nostri scopi è sufficiente anche una versione base, come, per esempio, Eclipse Galileo.
Sfortunatamente Eclipse non è in grado di compilare nativamente sorgenti per Android, ma è possibile scaricare un plugin gratuito, che permetterà di riconoscere la Tool-Chain di sviluppo Android installata nel sistema e configurarlo opportunamente. Questo plugin si chiama **ADT Plugin** ed è scaricabile dal sito di Android al seguente indirizzo *http://developer.android.com/sdk/eclipse-adt.html*, oppure più comodamente on-line all'interno dell'Ide, cliccando sulla voce di menu Help e poi cliccando su *Install New Software* e riempiendo i campi con l'in-

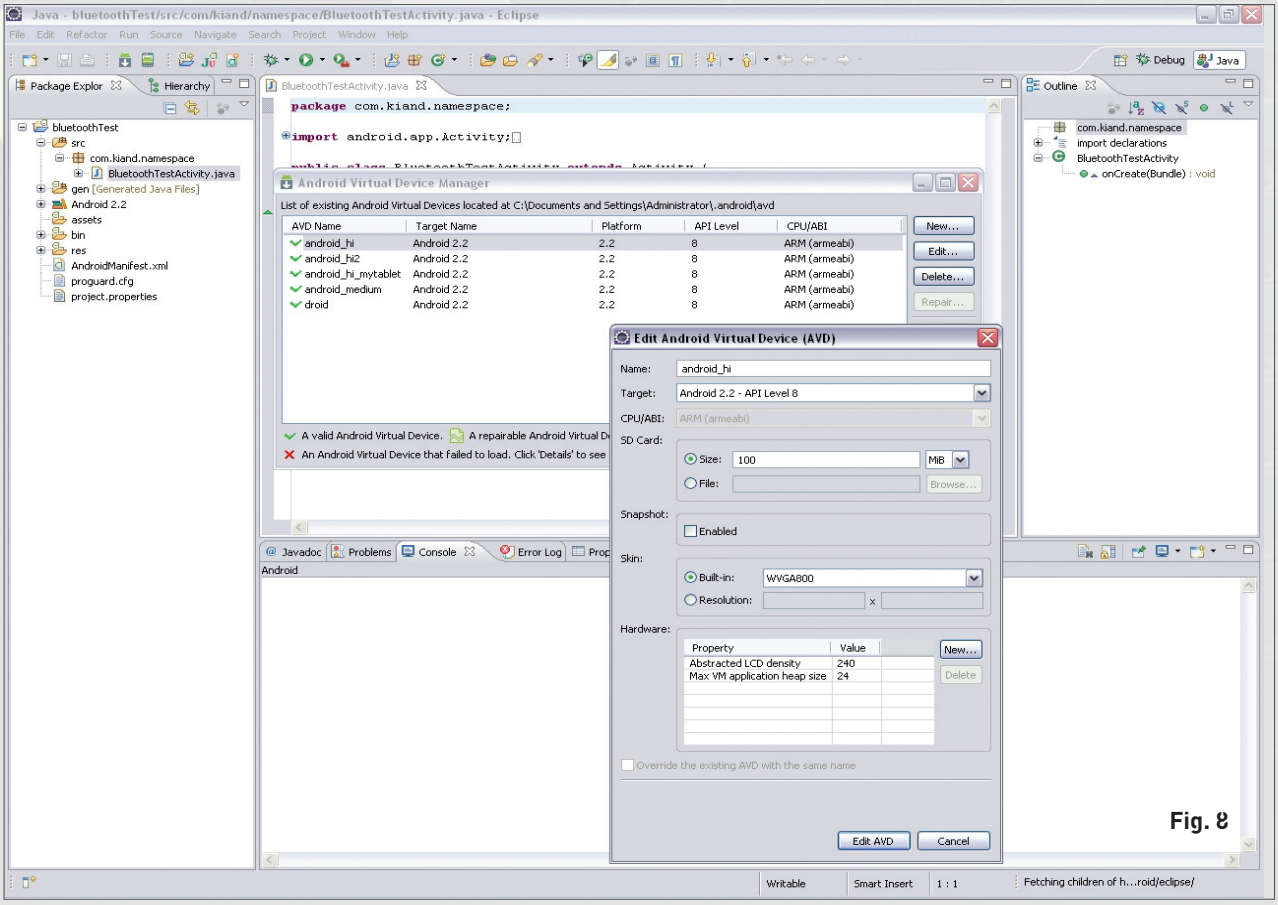


Fig. 8

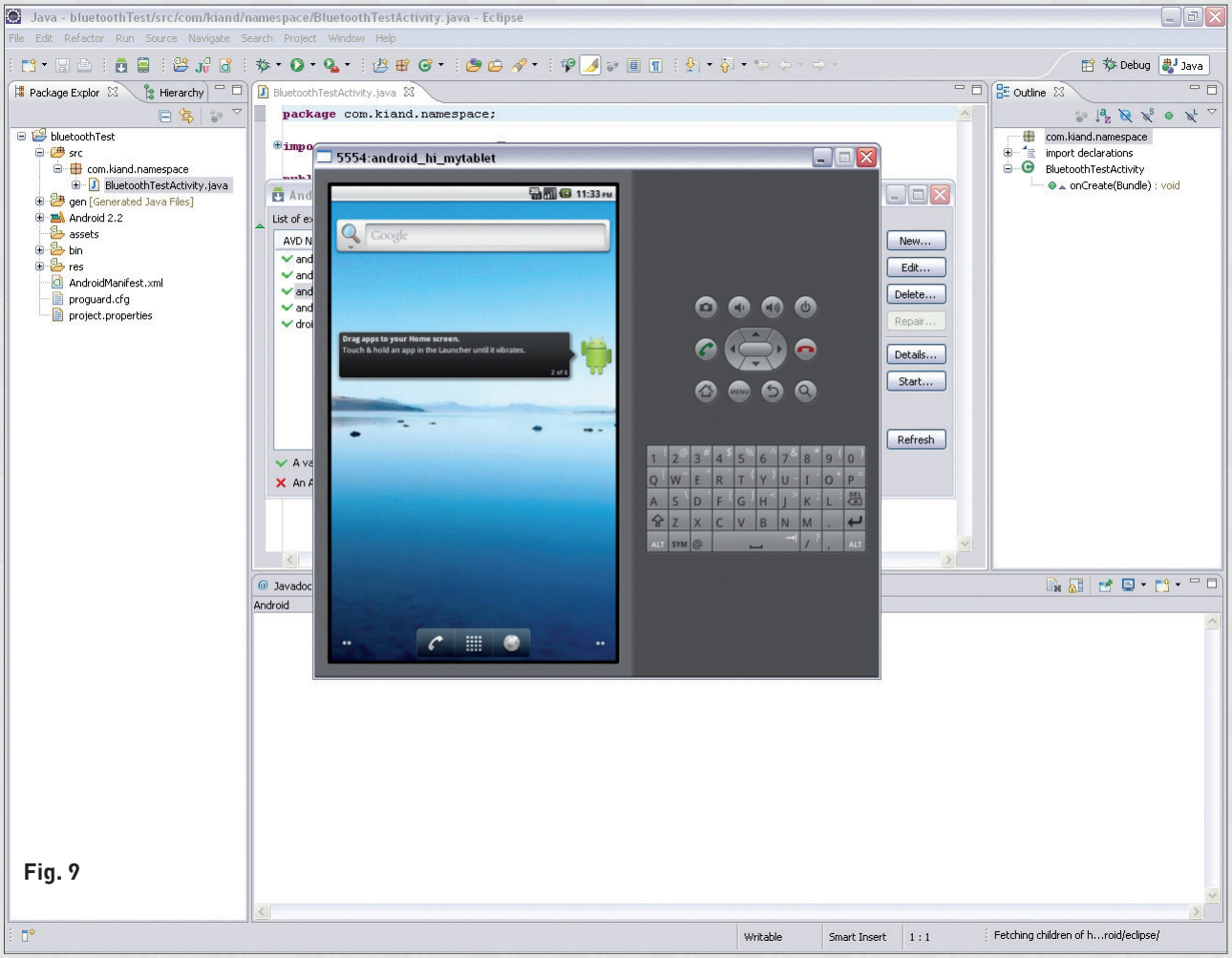


Fig. 9

dirizzo opportuno come indicato in **Fig. 5**. Al termine dell'installazione dovremo aprire la finestra di dialogo *Preferences* accessibile dalla voce di menu *Window* ed indicare nel campo *Android* in quale directory risiede il Platform SDK; otterremo un elenco delle librerie installate simile a quello in **Fig. 6**.
A questo punto possiamo creare un nuovo progetto Android come in **Fig. 7** e seguire i vari passi del wizard (che come potrete notare richiederanno l'inserimento degli stessi parametri visti nel caso di creazione da linea di comando) prima di eseguire la prima build cliccando col tasto destro sul nome del progetto nella vista *PackageExplorer* sulla parte sinistra dell' Ide e cliccando infine su *Build Project*.
Non ci soffermeremo in questa prima puntata nel descrivere tutte le funzionalità di Eclipse, ma come primo passo sarà sufficiente tenere d'occhio la "vista" *Problems*, nella parte bassa dell'Ide che ci darà informazioni se la build è andata a buon fine e nel caso contrario, ci darà indicazioni aggiuntive sugli errori ed eventuali warning.

L'emulatore Android

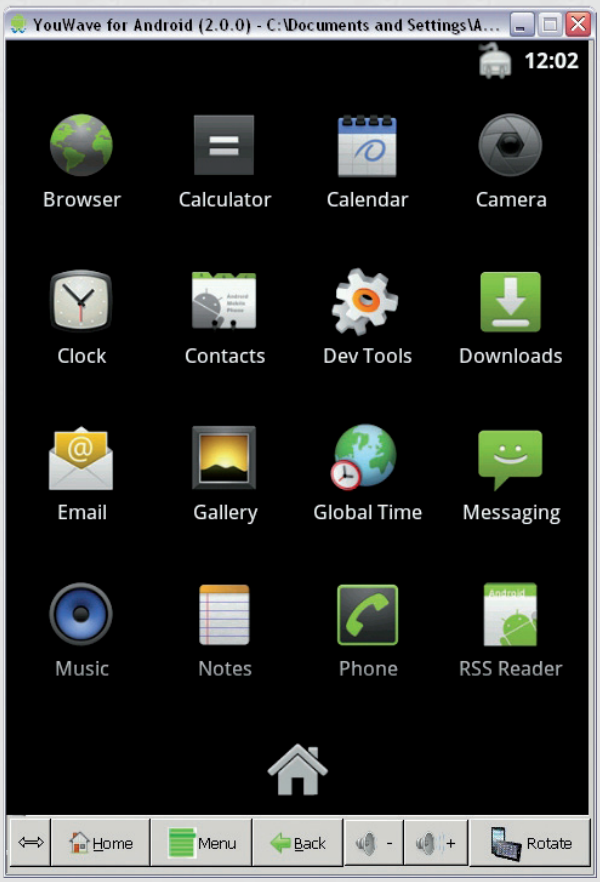
Rimandando maggiori dettagli di Eclipse alle puntate successive possiamo concludere analizzando un altro importante strumento che ci viene offerto dal pacchetto SDK Android appena installato, e cioè l'emulatore. Questo permetterà di testare l'applicazione e debuggarla sul PC senza dover disporre necessariamente di un dispositivo fisico e accelerando, così, i tempi di produzione.
Inoltre, offre un vantaggio non trascurabile su applicazioni Android destinate a funzionare su dispositivi dotati di caratteristiche hardware diverse tra di loro, basti pensare alla risoluzione e alle dimensioni degli schermi di differenti marche e modelli.
Se infatti, avessimo intenzione di pubblicare una nostra applicazione sul market Google

Play (il market store Android visitabile a https://play.google.com/), questa dovrà essere in grado di funzionare ed adattarsi su qualunque dispositivo e non potendo averli tutti a disposizione possiamo crearli a nostro piacimento sull'emulatore e testarli separatamente. Questo è possibile tramite l'eseguibile **AVD Manager** (Android Virtual Device) presente nella cartella installata (o direttamente da Eclipse) e visibile in **Fig. 8**. Dalla finestra principale possiamo creare nuovi dispositivi virtuali, ognuno con una diversa risoluzione ed associare loro una specifica versione delle librerie SDK (API) scelte tra quelle installate in precedenza (nel nostro caso Android2.2, API Level 8). Una volta creati quelli desiderati possiamo sceglierne uno ed avviarlo direttamente da questa finestra e dopo pochi minuti potremo navigare all'interno del nostro nuovo dispositivo usando il mouse come se fosse un touch e lanciando le varie applicazioni di sistema come su un vero smarphone Android (**Fig. 9**). Questo si rivela molto utile anche per prendere un po' di familiarità e dimestichezza col nuovo mondo Android, perchè, al di là della nostra applicazione, possiamo navigare nelle cartelle e nelle applicazioni di sistema come se fosse uno smarthone reale. Ovviamente l'emulatore potrà essere eseguito direttamente da ambiente Eclipse in seguito ad una nuova compilazione cliccando col tasto destro del mouse sul nome e successivamente su *Run As – Android Application*. Seguirà una finestra in cui potremo scegliere quale dispositivo vogliamo emulare tra quelli creati in precedenza e la nostra applicazione verrà eseguita direttamente sull'emulatore. Da notare come non sia il caso di chiudere la finestra dell'emulatore al termine delle nostre prove, ma anzi, conviene tenerla sempre aperta in modo da ridurre i tempi di caricamento; al successivo conseguimento della build, il sistema troverà l'emulatore già attivato e l'applicazione verrà caricata più rapidamente per essere eseguita e testata nuovamente.
Con le ultime versioni del Platform SDK Android, sono stati fissati alcuni bug che rendevano i tempi di caricamento del dispositivo virtuale veramente lunghi e tali da renderne impraticabile il test ed il debug. Tuttavia, su alcune macchine piuttosto lente, potrebbero esserci ancora problemi di questo tipo per cui consigliamo di installare un emulatore di terze parti che è molto usato e soprattutto molto più veloce. Si chiama **YouWave Android**, è scaricabile da questo sito *http://youwave.com* e, anche se non è gratuito, non ha un costo proibitivo ed esiste la possibilità di provarlo per un tempo limitato di sette giorni (**Fig. 10**). Copiando l'applicazione generata da Eclipse, (un file .apk che avremo modo di analizzare in seguito) in una specifica cartella di YouWave saremo in grado di installarla e provarla su questo emulatore in tempi veramenti brevi.

Conclusioni

Ora abbiamo tutti gli strumenti per sviluppare su Android, ma per il momento ci fermiamo qui, permettendovi di testare il nuovo ambiente e rimandando alla prossima puntata i primi esempi di codice java e le prime prove sul dispositivo fisico. ■

Fig. 10



Riprendiamo il corso di Android soffermandoci sull'ambiente di sviluppo Eclipse per emulare la nostra prima applicazione ed eseguirla su un dispositivo reale. Seconda Puntata.

di Andrea Chiappori

2

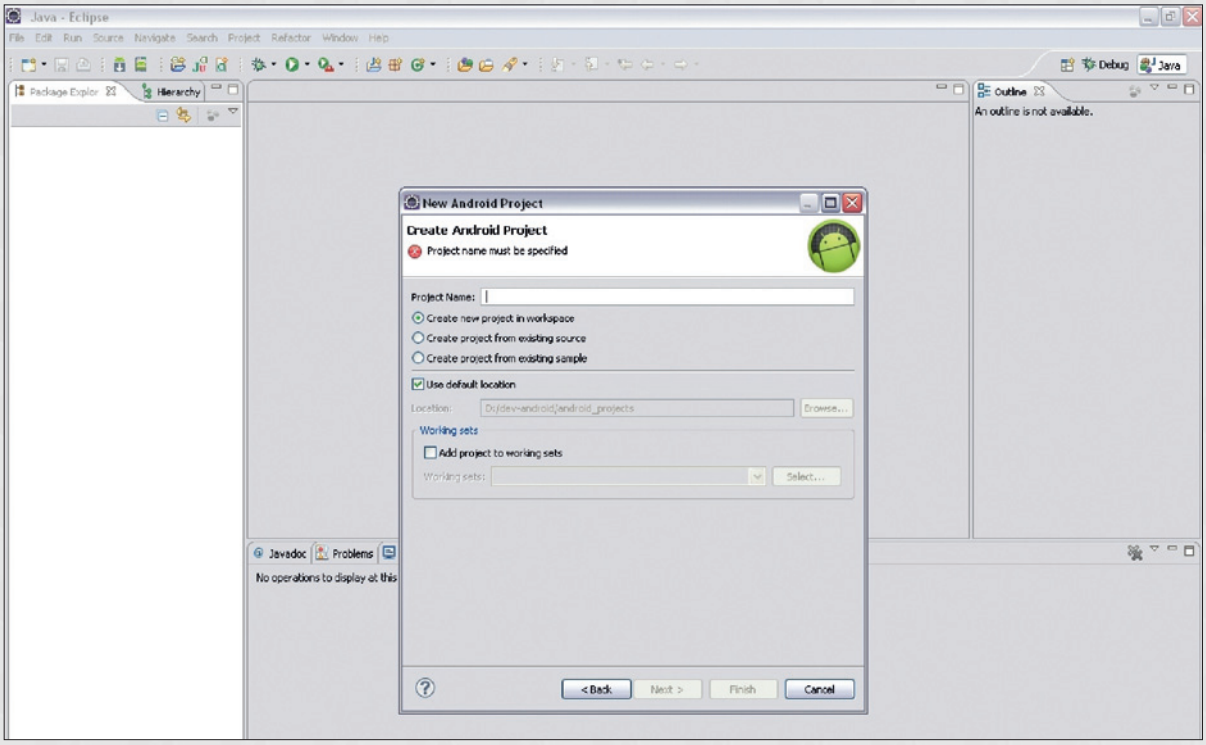
PROGRAMMIAMO CON ANDROID

Dopo l'introduzione al mondo Android della puntata precedente siamo pronti a programmare la nostra prima applicazione sfruttando l'ambiente di sviluppo Eclipse presentato in precedenza ed analizzando la struttura di un progetto nei suoi dettagli. Saremo in grado di testare il nostro prodotto sull'emulatore fornito dal plugin ADT di Android e finalmente di vederlo in azione sul nostro dispositivo. Sarà ancora un progetto senza particolari funzionalità pratiche e privo di interfaccia grafica, ma ci permetterà di comprendere il processo completo per programmare, testare e pubblicare una nostra futura applicazione sul market Android.

Eclipse

Eclipse è un ambiente di sviluppo integrato (IDE) nato da IBM e diventato successivamente Open Source, è sviluppato in Java; essendo un ambiente multi-linguaggio è possibile sviluppare applicazioni programmando anche in C/C++, a patto di installare i plugin necessari. È infatti bene ricordare che Eclipse, da solo, ci consente di sviluppare programmi in Java, ma non basta a creare un progetto Android. Nel nostro caso programmeremo in Java, ma avremo comunque bisogno del plugin ADT (Android Developers Tools) già installato nella puntata precedente, per permettere ad Eclipse di creare un nostro progetto e di fornirci strumenti specifici come l'emulatore, i driver, le librerie e un insieme di esempi.
Esistono altri ambienti di sviluppo per programmare con Android (come ad esempio IntellijIDEA e NetBeans per citarne alcuni) , ma abbiamo scelto di utilizzare Eclipse perchè è il più usato in ambito di applicazioni embedded;

Fig. 1



molti dei lettori lo avranno già utilizzato e chi si ritroverà ad utilizzarlo anche per altre piattaforme che non siano Android, avrà più dimestichezza nel navigare al suo interno.
D'altro canto questo non vuole essere un corso specializzato su Eclipse, che inizialmente potrà apparire un pò ostico, per la presenza di molte funzioni che non analizzeremo in queste pagine, ma una base dalla quale partire per avvicinarci alla programmazione di sistemi Android.
A questo punto possiamo iniziare.

Il nostro primo progetto

Una volta aperto Eclipse possiamo creare il nostro primo progetto scegliendo File-NewProject dal menu, cliccando su Android Project si aprirà una finestra come in **Fig. 1**, da dove inizierà la procedura guidata nella quale inseriremo i parametri già visti nella puntata precedente.
Ricordiamoci di scegliere, nella schermata di **Fig. 2**, le librerie opportune in base alla versione Android installata sul dispositivo finale; nel nostro esempio abbiamo scelto le 2.2 perchè utilizzeremo uno smartphone con Android Froyo (ovviamente funzionerà anche su versioni successive).
Terminata questa fase Eclipse avrà creato per noi lo scheletro di un progetto che visualizzerà il classico "Hello Word" sullo schermo in modalità console.

Struttura del progetto

Prima di iniziare a modificare ed aggiungere il nostro codice è bene dare uno sguardo alla struttura del progetto, che risulta organizzato in tante sotto-directory.
Queste sono visualizzate nella vista (o view) a sinistra dello schermo e rispecchiano le directory del file system che potrete trovare navigando con Esplora Risorse nella cartella del progetto appena creata.
Possiamo vedere come si presenta in **Fig. 3** analizzandole brevemente qui di seguito:
La directory src contiene i file sorgenti con estensione .java. Al momento della creazione

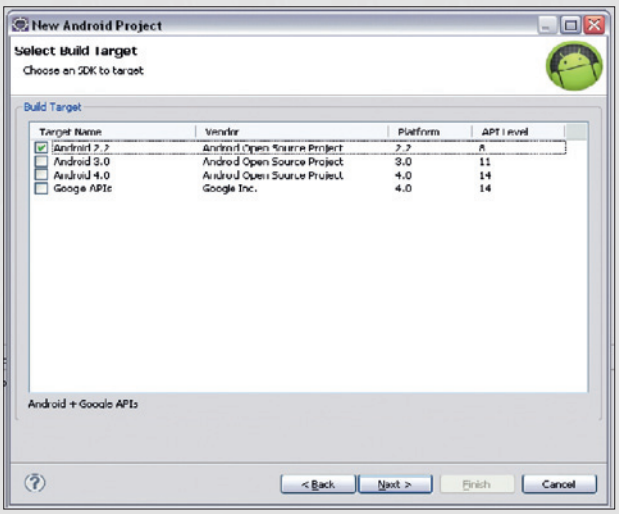


Fig. 2

Fig. 3

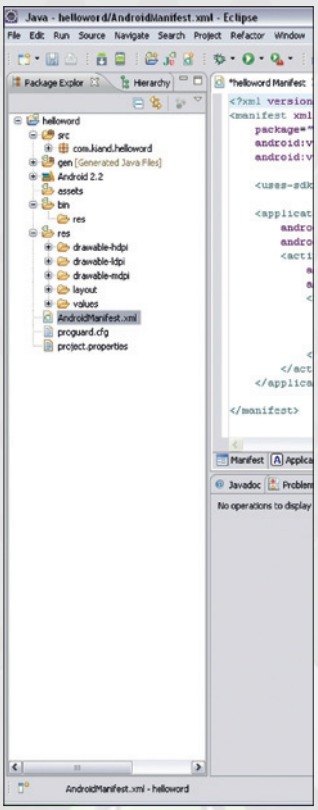
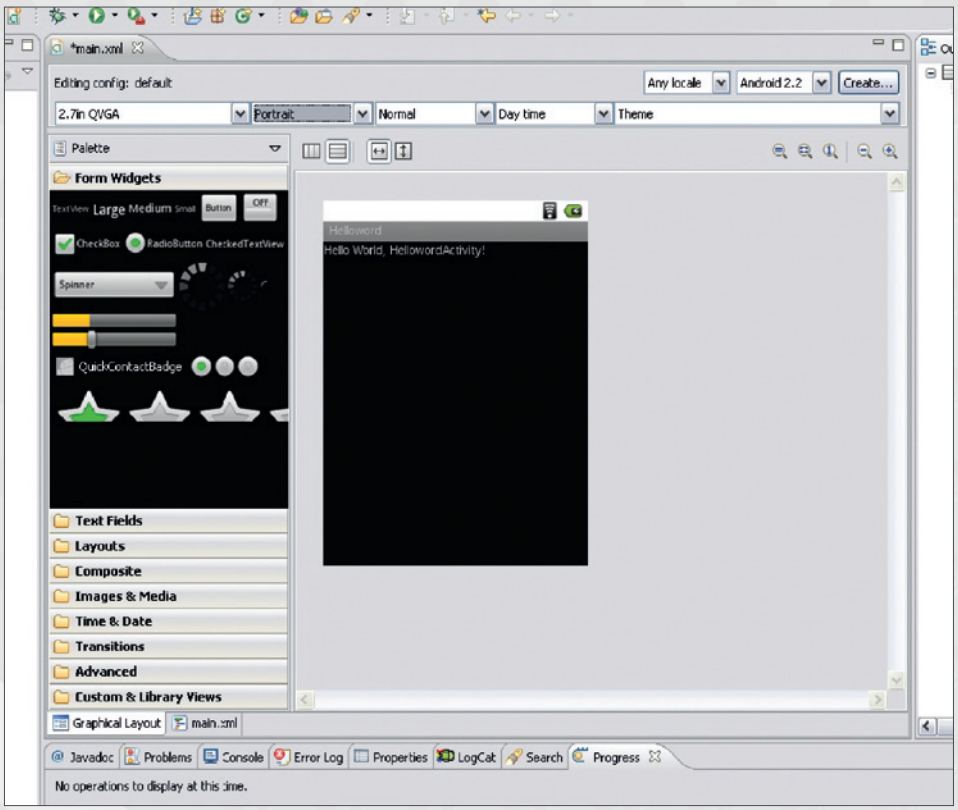


Fig. 4



del progetto conterrà un solo file con l'unica classe principale dell'applicazione.
Ogni file con estensione .java presente all'interno di questa cartella verrà compilato all' esecuzione del comando build, per cui se volessimo aggiungere le nostre classi basterà creare un file all'interno della stessa cartella ed aggiornare la vista premendo F5 o col comando Refresh. Questa cartella conterrà il codice del progetto ma, come vedremo ora, da sola non basta per generare il prodotto finale.
La directory gen conterrà invece file autogenerati dal progetto durante la compilazione; questi contengono riferimenti alle risorse del progetto come ad esempio bottoni, immagini, file audio e stringhe, ma dal momento che non dobbiamo modificarli, possiamo temporaneamente trascurarla.
La directory Android 2.2 è in realtà un collegamento ad un file precompilato .jar che non si trova all'interno della cartella del progetto, ma nelle cartelle create al momento dell'installazione del plugin ADT e contiene tutte le librerie che ci permettono di sviluppare con Android.
Eclipse ci permette di visualizzarle tutte ed elencare anche le funzioni che ognuna di esse presenta. Questa caratteristica, anche se non è indispensabile, può tornare utile soprattutto per i più curiosi, perchè fornisce informazioni su come prendere il controllo del nostro dispositivo e può suggerirci nuovi spunti per le nostre future applicazioni.
La directory bin contiene i file binari che vengono utilizzati dall'applicazione, nonchè l'applicazione stessa in formato nativo (.dex) ed il prodotto finale con estensione .apk. Ricordiamo che il risultato ultimo della nostra compilazione sarà un file .apk che è un file compresso contenente al suo interno le risorse dell'applicazione, l'eseguibile nativo per Android (.dex) e qualche altro file che sarà possibile vedere rinominando il file apk con estensione .zip. Il file con estensione .apk sarà il solo che andremo ad uploadare sul market android quando avremo terminato il nostro progetto.
La directory res contiene le risorse utilizzate dal nostro progetto; questa cartella è a sua volta suddivisa in altre sotto-directory in base al tipo della risorsa specifica. È bene analizzare meglio le cartelle ed i file contenuti in questa directory, perchè assumono un aspetto considerevole per quanto riguarda l'aspetto, lo stile e l'interfaccia grafica della nostra applicazione e che quindi non va tralasciata perchè sarà il primo diretto contatto con l'utente finale. Analizziamo perciò, di

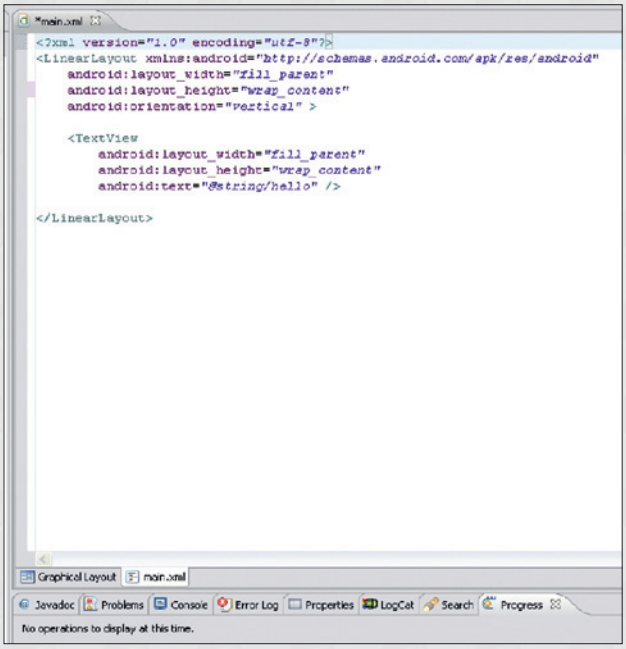


Fig. 5

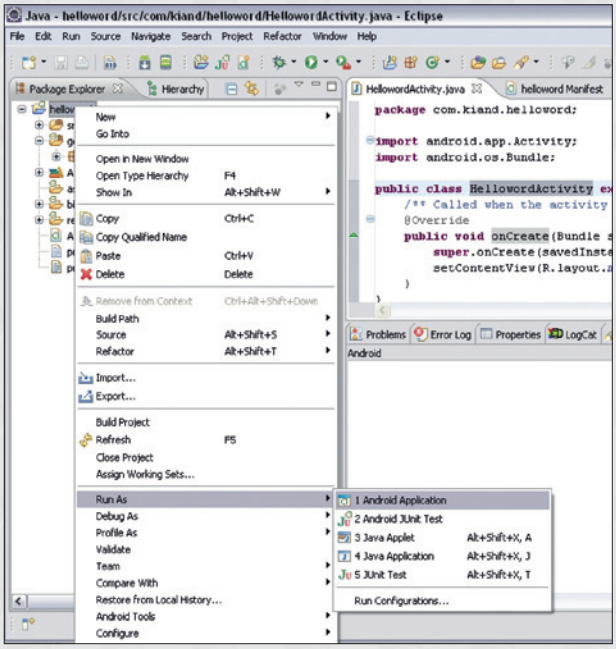
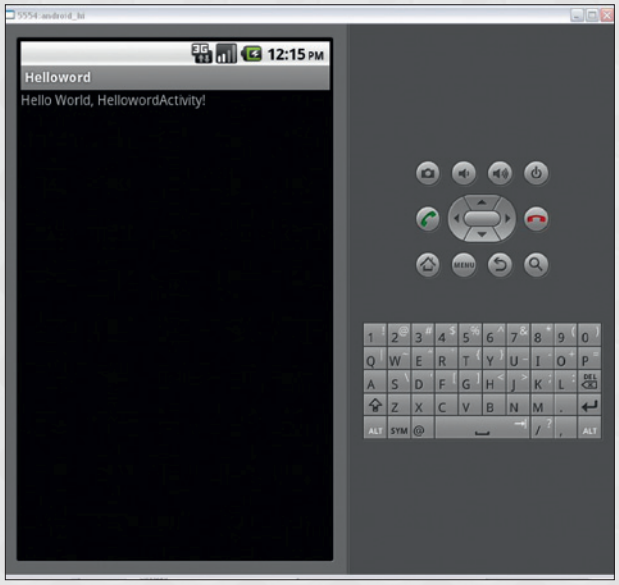


Fig. 6

seguito le sotto-directory della cartella res. La sotto-directory values presenta al suo interno un file .xml strings.xml, che contiene le stringhe utilizzate dalla nostra applicazione. Ad esempio nel progetto appena creato questo file contiene la stringa "Hello World", noi potremmo naturalmente aggiungere altre stringhe e vedremo in seguito in che modo fare riferimento ad esse per poterle usare richiamandole dal codice dell'applicazione. La directory values potrebbe contenere anche più di un file di stringhe, possiamo infatti pensare ad una applicazione che supporti più linguaggi per le diverse nazionalità.
La sotto-directory layout contiene file .xml che rappresentano l'interfaccia grafica della nostra applicazione. Al momento ne vediamo solo uno, (main.xml) ed è molto scarno, ma in seguito la completeremo a seconda delle nostre esigenze aggiungendo i bottoni ed i controlli che più ci torneranno utili. Se alla nostra applicazione servisse anche una finestra di dialogo, piuttosto che una finestra per settare le opzioni, o ancora un menu, basterà aggiungere altri file .xml, ognuno con la propria veste grafica che richiameremo direttamente dal codice come vedremo più avanti. Cliccando su questi file si aprirà sulla vista principale l'aspetto della nostra form (visibile in **Fig. 4**) in cui potremmo aggiungere tutti i controlli possibili (presenti sulla sinistra) semplicemente trascinandoli sopra. Eclipse, in base alle nostre azioni, genererà automaticamente il file xml (nel nostro esempio main.xml) che potremo visualizzare cliccando sul tab posizionato in basso alla finestra con scritto il nome del file, come in **Fig. 5**.
Nulla ci vieta, anzi la maggior parte delle volte è anche estremamente consigliabile, modificare l'aspetto della nostra applicazione inserendo controlli e settando le loro proprietà (colore, posizione, stile..) scrivendo direttamente nel file xml. Questo presuppone una maggiore conoscenza delle librerie Android, ma di sicuro permette un maggior controllo e chiarezza quando dovremo modificare o aggiungere altri elementi. Rimane comunque il vantaggio che dopo aver scritto manualmente il file .xml potremmo vederne il risultato tornando in modalità grafica, cliccando sul tab Graphical Layout. Prossimamente vedremo più in dettaglio questo aspetto.
Le sotto-directory drawable-hdpi, drawable-mdpi, drawable-ldpi contengono le immagini utilizzate dall'applicazione e sono più di una per permettere l'adattamento a dispositivi con diverse dimensioni dello schermo. Al momento della creazione del progetto queste cartelle conterranno già la classica icona Android che rappresenterà la nostra applicazione. In ognuna di queste cartelle troveremo un file immagine in formato png, chiamato con lo stesso nome, ma con risoluzione diversa in base alla densità dello schermo. In particolare nella cartella drawable-hdpi troveremo l'immagine dell'icona maggiormente definita e pensata per schermi ad alta risoluzione (solitamente 480x640), nella cartella drawable-mdpi avremo la stessa icona, ma meno

Fig. 7



definita, per schermi di media risoluzione (320x480) e nella cartella drawable-ldp l'icona sarà per schermi a bassa risoluzione come negli smartphone più piccoli (240x320).
Ovviamente quando andremo ad inserire le nostre eventuali risorse grafiche dovremo preoccuparci di generare la stessa immagine, ma con risoluzione diversa ed inserirla nella corrispondente cartella. La risoluzione deve rispettare certi canoni che non svilupperemo in questa sede, ma che si trovano dettagliatamente spiegati in un documento fornito dalla comunità di Android al seguente indirizzo: *http://developer.android.com/guide/practices/screens_support.html*
La cartella assets può contenere dati e risorse di ogni genere, come immagini, audio, testo, file binari; solitamente non è molto usata, ma esiste una funzione di libreria (getAssets()) con la quale si può accedere a questi file ed utilizzarli per i propri scopi all'interno della nostra applicazione.
Infine, concludendo con la struttura dei file di progetto, troveremo, fuori da ogni directory, il file manifest.xml, anch'esso editabile sia graficamente sia manualmente e contenente informazioni specifiche riferite all'applicazione, come il nome vero e proprio, il numero di versione, la funzione di partenza, lo stile grafico, l'orientazione e l'elenco dei permessi. È infatti buona norma rendere disponibile all'utilizzatore finale le informazioni su quello che potrà utilizzare la nostra applicazione. Se ad esempio pensiamo di dover utilizzare il modulo bluetooth o accedere alla rete, piuttosto che ai file interni alla sd-card dovremo specificarlo opportunamente in modo che al momento dell'installazione venga segnalato all'utente finale che potrà, così, scegliere o meno di proseguire.

Uno sguardo al codice

Abbiamo già visto nella puntata precedente come Android fornisca insieme al plugin ADT anche l'emulatore per poter testare la propria applicazione, prima di scaricarla direttamente sul dispositivo reale.
A questo punto siamo in grado di generare la nostra prima applicazione apk e provarla prima sull'emulatore ed infine sul dispositivo Android. Prima di lanciare la compilazione dell'applicazione, possiamo dare uno sguardo al codice automaticamente generato da Eclipse e prendere confidenza con i comandi basilari dell' interfaccia Eclipse.
Come visibile in **Fig. 6**, il codice risulta piuttosto semplice e può servire per introdurre alcune nozioni che vedremo più in dettaglio nelle prossime puntate.
In questa applicazione basilare, la classe è una sola (HellowordActivity) e tramite la keyword extends deriva dalla classe base Activity (da qui il motivo dell'importazione import android.app.Activity). L'Activity in Android può essere pensata come una schermata grafica o "screen" al quale può essere associato un layout che altro non è che un file xml presente nella sotto directory layout, esaminata in precedenza e che conterrà l'interfaccia grafica vera e propria. Questa associazione avviene tramite la funzione setContentView() al quale viene passato come argomento R.layout.main che rappresenta proprio il file xml main.xml. Più in dettaglio, per entrare nella filosofia Android, R è una classe autogenerata da Eclipse e la possiamo trovare nel file R.java all'interno della directory gen; nella classe R sono contenute altre classi che corrispondono alle risorse dell'applicazione e che contengono un valore univoco (Id) dei singoli elementi (immagini, audio, stringhe, file xml). Per cui con R.layout.main viene proprio specificato di utilizzare la risorsa main.xml contenuta nella directory layout all'interno della cartella di risorse res. Da notare come occorra non specificare l'estensione xml per il file main.xml. Una applicazione Android potrà contenere più di una activity ed ognuna potrà

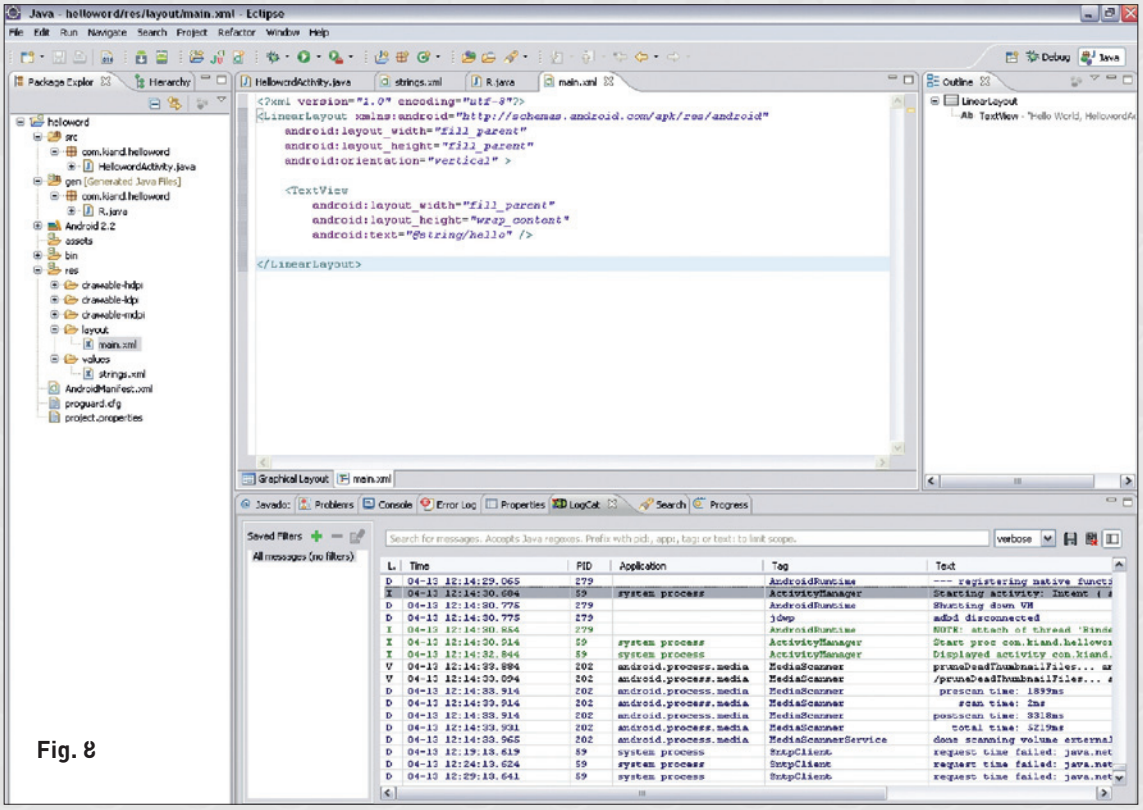


Fig. 8

essere associata ad un file di risorse diverso in modo da presentare un layout diverso e quindi una interfaccia grafica diversa.
La funzione setContentView() deve essere chiamata una volta sola ed al momento della partenza dell'applicazione, pertanto viene ridefinito, tramite override, il metodo onCreate() della classe base (Activity) al quale viene aggiunta proprio la funzione setContentView(), dopo la classica onCreate() della classe base. Con la keyword super viene, infatti, indicato che la onCreate() da utilizzare è quella della classe base Activity.

Test su emulatore

Ora, per compilare, possiamo lanciare la build cliccando col tasto destro sul progetto e poi su Build Project sul menu; all'interno della cartella bin troveremo il file helloword.apk appena generato. Nel caso la compilazione non fosse andata a buon fine potremmo leggere gli errori o i warnings nella finestra situata nella parte inferiore di Eclipse ed in particolare sulla vista chiamata Problems dove vengono indicate le righe interessate dagli errori.
A questo punto possiamo provare l'applicazione con l'emulatore, cliccando col tasto destro sul progetto e poi su Run As e Android Application come mostrato in **Fig. 6**.
La prima volta che viene avviato l'emulatore potranno trascorrere svariati secondi, ma saremo in grado di seguire i vari passi (ed eventuali errori) del processo tramite la vista Console di Eclipse, al termine dei quali otterremo una finestra come in **Fig. 7**.
L'applicazione è ovviamente molto semplice, ma può servire per comprendere meglio alcuni meccanismi di Android. Da notare come la stringa "Hello world, HelloWordActivity!" che compare nella parte alta dello schermo non sia presente nel codice del file HellowordActivity.java esaminato in precedenza. La spiegazione è che la stringa è inserita nel file di risorse strings.xml all'interno della cartella res ed è identificata con il nome hello, questo stesso nome lo ritroviamo nel file di risorse main.xml che rappresenta il nostro layout. Questo file è stato autogenerato presentando già al suo interno un primo semplice controllo, un TextView che presenta, tra le sue proprietà, la stringa iniziale che è appunto hello (**Fig. 8**).
Durante la fase di emulazione è anche possibile visualizzare i messaggi di log che vengono inviati dal sistema Android verso l'esterno, tramite la vista LogCat di Eclipse, situata nella parte inferiore della finestra. Possiamo però, come vedremo nelle puntate successive, inserire alcune righe di log personalizzato

Fig. 9

all'interno della nostra applicazione in modo da seguirne il flusso tramite questa finestra.

Test su dispositivo Android

Per completare i passaggi possiamo ora analizzare la procedura per scaricare finalmente la nostra app su un dispositivo reale.
Occorrerà prima di tutto aver installato i driver USB del dispositivo che solitamente sono forniti al momento dell'acquisto del dispositivo Android, in modo che il vostro PC lo riconosca correttamente una volta collegato con il cavetto USB.
Poi sarà necessario impostare il dispositivo in modo tale da permettere lo scarico ed il debug dell'applicazione. Per fare questo bisognerà entrare nel menu Impostazioni di Android, selezionare la voce Applicazioni, poi Sviluppo e spuntare la casella Debug USB come in **Fig. 9**. Comparirà allora il simbolino Android sulla parte alta del vostro dispositivo che sarà così in grado di colloquiare tramite Eclipse.
Al prossimo avvio di Eclipse, infatti, rilanciando l'applicazione tramite Run As e Android Application da tasto destro la vista Console vi informerà sullo scarico e in pochi secondi vedrete la vostra applicazione sul dispositivo reale.
Naturalmente anche in questa fase nella vista LogCat saranno presenti i messaggi di log del dispositivo.

Pubblicare App su Market Android

Non è di sicuro lo scopo principale di questo articolo anche perchè le applicazioni che vedremo nelle prossime puntate saranno sempre abbinate ai nostri dispositivi a microcontrollore, ma se volessimo pubblicare le nostre app sul Google Play (il market di Android) possiamo farlo in brevi passi.
Innanzi tutto occorre registrarsi al seguente url *https://market.android.com/publish* come programmatore Android e pagare 25 dollari.

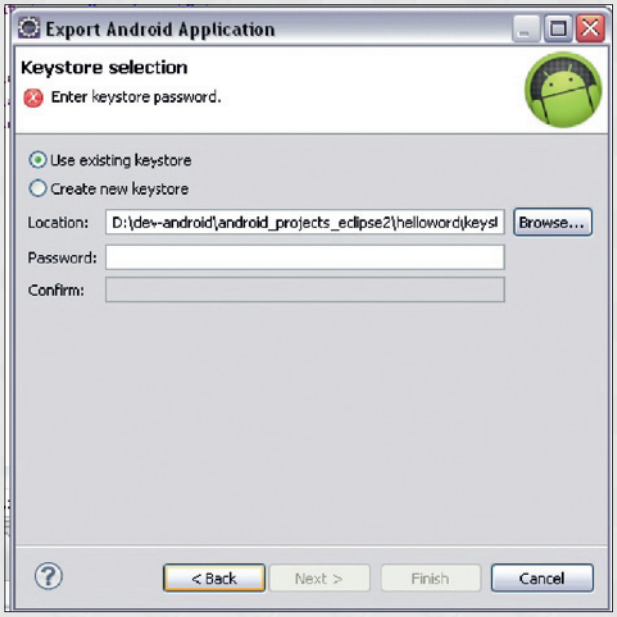


Fig. 10

Saremo cosi' forniti di una chiave che potremo utilizzare per firmare le nostre applicazioni.
Sempre da Eclipse, col tasto destro sul nome del progetto scegliamo Android Tools e poi Export Signed Application Package. Si aprirà una finestra come in **Fig. 10** in cui potremo scegliere se utilizzare una nuova chiave oppure usarne una già creata in precedenza.
Una volta completati questi passi avremo il nostro pacchetto apk firmato che potremo caricare su Google Play tramite la nostra pagina internet dell'account Google. Da questa pagina dovremo inserire almeno due screenshot della nostra applicazione, una breve descrizione e una icona ad alta risoluzione che la rappresenti. Sono inoltre disponibili a questo indirizzo *https://support.google.com/googleplay/android-developer* una serie di linee guida da seguire per rispettare lo stile Android e rivolte esplicitamente a sviluppatori Android.
Infine dopo alcune ore dal nostro upload potremo trovare la nostra applicazione direttamente sul market Android (Google Play).

Conclusioni

Abbiamo ora tutte le basi per poter iniziare a prendere confidenza col mondo Android e creare qualcosa di nuovo.
Nella prossima puntata analizzeremo il codice di una prima applicazione che si interfaccerà con dispositivi elettronici esterni, prendendo spunto per esaminare alcuni controlli grafici di Android e la loro interazione con l'utente.

Realizziamo una prima applicazione Android in grado di controllare dispositivi elettronici remoti utilizzando il protocollo Bluetooth e soffermandoci in questa puntata sulla interfaccia utente. Terza Puntata.

di Andrea Chiappori

3

PROGRAMMIAMO CON ANDROID

Abbiamo visto, nelle puntate precedenti, come realizzare un semplice progetto Android, ma ancora privo di controlli visuali ed interfaccia utente. In queste pagine verranno mostrati i primi passi con cui daremo vita ad una applicazione Android completa ed in grado di comunicare con il mondo esterno. Amplieremo, pertanto, la trattazione precedente analizzando l'inserimento di controlli grafici che ci consentiranno di variare la luminosità di una striscia led RGB (o qualsiasi altro dispositivo luminoso) collegata su una piattaforma Arduino con lo shield RGB-Shield presentato nella rivista numero 159.
Per realizzare tutto questo senza fili, utilizzeremo il protocollo Bluetooth, visto che, ormai, tutti i dispositivi ne sono dotati, e lo useremo per comunicare con un altro modulo Bluetooth (questa volta esterno), collegato ad Arduino tramite la porta seriale. Lo schema di principio è rappresentato in **Fig. 1**. Prima di analizzare l'hardware, occupiamoci del software e, più in dettaglio, dell'interfaccia grafica che verrà visualizzata sul dispositivo Android.

Il Software

Android gestisce l'interfaccia grafica tramite uno o più file xml, mediante i quali possiamo definire quello che viene chiamato layout, ovvero la disposizione in cui andremo ad inserire i controlli necessari per la nostra applicazione.
Al momento della creazione di un progetto abbiamo già a disposizione un layout basilare che troviamo all'interno della sottocartella res/layout come già visto nella puntata precedente.
Questo file, però, sarà piuttosto scarno perchè presenta solamente il controllo TextView, ovvero una casella di testo (per chi programma con C#

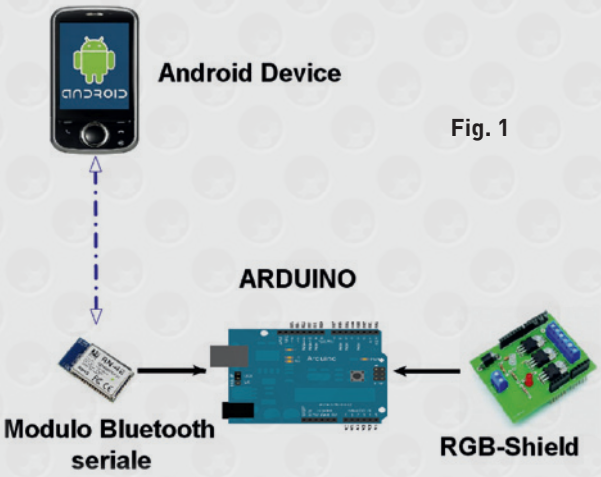


Fig. 1

corrisponde al controllo Label) su cui viene scritta una stringa; adesso siamo in grado di arricchirlo con controlli grafici come bottoni, slide, caselle di testo e tanti altri che ci permetteranno di interagire col modulo Bluetooth esterno. Questi controlli grafici sono chiamati viste (view) in quanto la classe base di ogni controllo è appunto la classe View.
Nella cartella *layout* possiamo inserire anche più di un file xml e ad ognuno di questi corrisponderà una schermata grafica con le corrispondenti view da noi inserite; in questo modo, qualora la nostra applicazione richiedesse altre finestre supplementari - come una *dialog-box*, una schermata iniziale o una qualsiasi altra finestra - potremo caricarla al momento opportuno dal codice. Vediamo come nel **Listato 1** dove viene caricata la pagina principale. Al momento della creazione dell'applicazione (classe *Activity*) viene chiamato il metodo onCreate() che a sua volta utilizza *setContentView()* passandogli il nome del file di *layout* che verrà usato e che visualizzerà l'interfaccia grafica contenuta al suo interno.
Ricordiamo, come già visto precedentemente, che la stringa *R.layout* rappresenta il percorso della risorsa all' interno del progetto (R identifica la cartella *res* e *layout* la sua sotto cartella) e *main* è appunto, il nome del file xml privato della sua estensione. Nulla vieta di chiamarlo con altri nomi. Appartengono al *layout* (e quindi andranno inseriti in questa stessa sottocartella) anche i file xml per la gestione dei menu, ovvero quella schermata a comparsa che viene visualizzata premendo il tasto di sistema. Vedremo successivamente come sarà gestita. Per semplicità inzieremo il nostro progetto con un unico file xml.

Listato 1

```
@Override
        public void onCreate(Bundle savedInstanceState)
        {
                super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        …
}
```

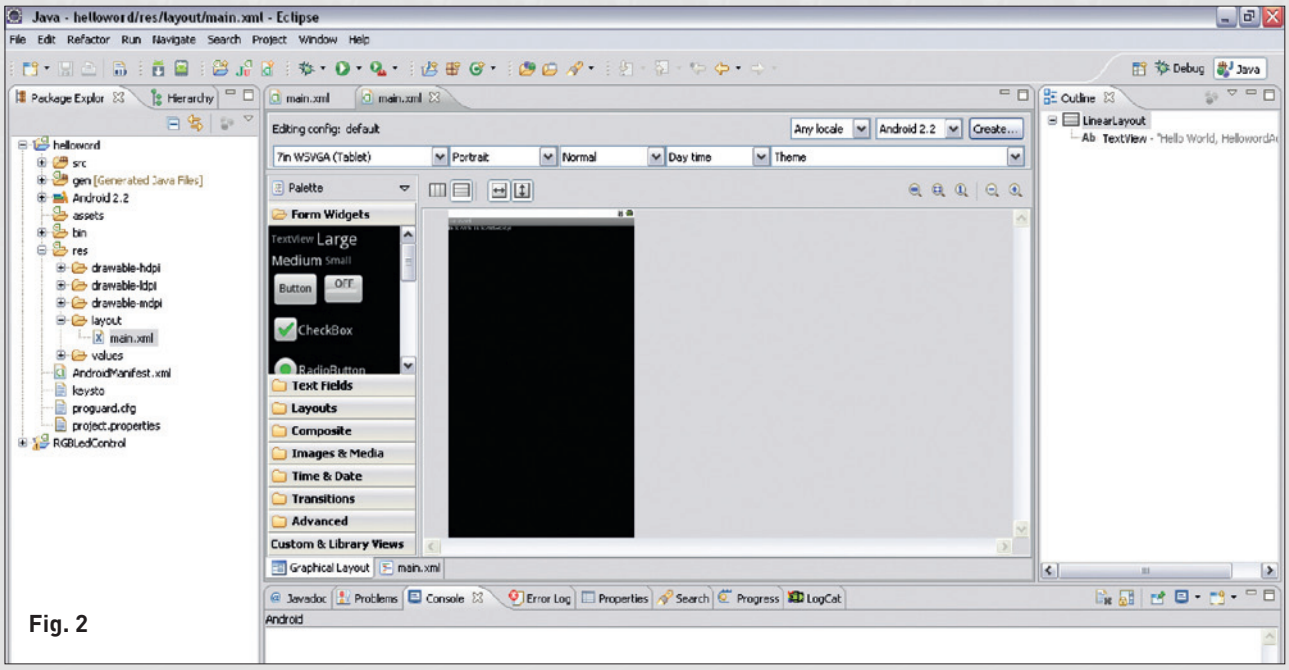


Fig. 2

Listato 2

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
                              android:layout_width="fill_parent"
                              android:layout_height="fill_parent"
                              android:orientation="vertical" >

    <TextView
        android:id="@+id/text01"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="34"
        android:textColor="#00F"
        android:text="RGB LED Control" />

</LinearLayout>
```

Interfaccia grafica

Vediamo ora come inserire i nostri controlli (o viste) nel file main.xml che costituirà il corpo principale dell'interfaccia grafica.
Chi è abituato a sviluppare applicazioni grafiche con Visual Studio (ad esempio con C#) si aspetterà un sistema analogamente semplice per inserire i vari controlli, ovvero scegliendoli e trascinandoli nell'area voluta dello schermo. In realtà è così anche con Eclipse, ma non proprio altrettanto semplice ed occorrono, perciò, alcune precisazioni.
Aprendo un file di *layout* comparirà una finestra come quella in **Fig. 2**, a questo punto potremmo scegliere di visualizzarlo in formato testo (cliccando sul tab *main.xml*) o in formato grafico (selezionando il tab *Graphical Layout*). In quest'ultimo formato possiamo selezionare i controlli desiderati sulla sinistra e trascinarli nella schermata centrale, ma questa operazione può portare, alla lunga, ad errori di posizionamento ed è suscettibile di continui aggiustamenti e correzioni, pertanto consigliamo questa modalità solo una volta divenuti più esperti. Utilizzando, invece, la modalità testuale per inserire i controlli, saremo costretti a capire la struttura xml che c'è dietro ad ogni interfaccia e, a parte le inevitabili piccole difficoltà iniziali, saremo in grado di controllare ogni sua parte e di modificarla con facilità quando se ne presenterà l'occasione.
Inoltre queste due modalità (grafica e testuale) vengono aggiornate insieme e costantemente da Eclipse, così se inseriamo un controllo agendo sul file xml, potremmo tornare sulla finestra *Graphical Layout* per valutare il risultato visivamente ed eventualmente correggere o modificare, senza nemmeno dover ricompilare il progetto continuamente. Una volta capiti i meccanismi che regolano la scrittura del *layout* potremmo impadronirci anche della sola modalità grafica, ma andiamo per gradi.
Innanzitutto è bene sapere che esistono diverse modalità con cui generare il proprio *layout*; il layout più usato è quello lineare (*LinearLa-*

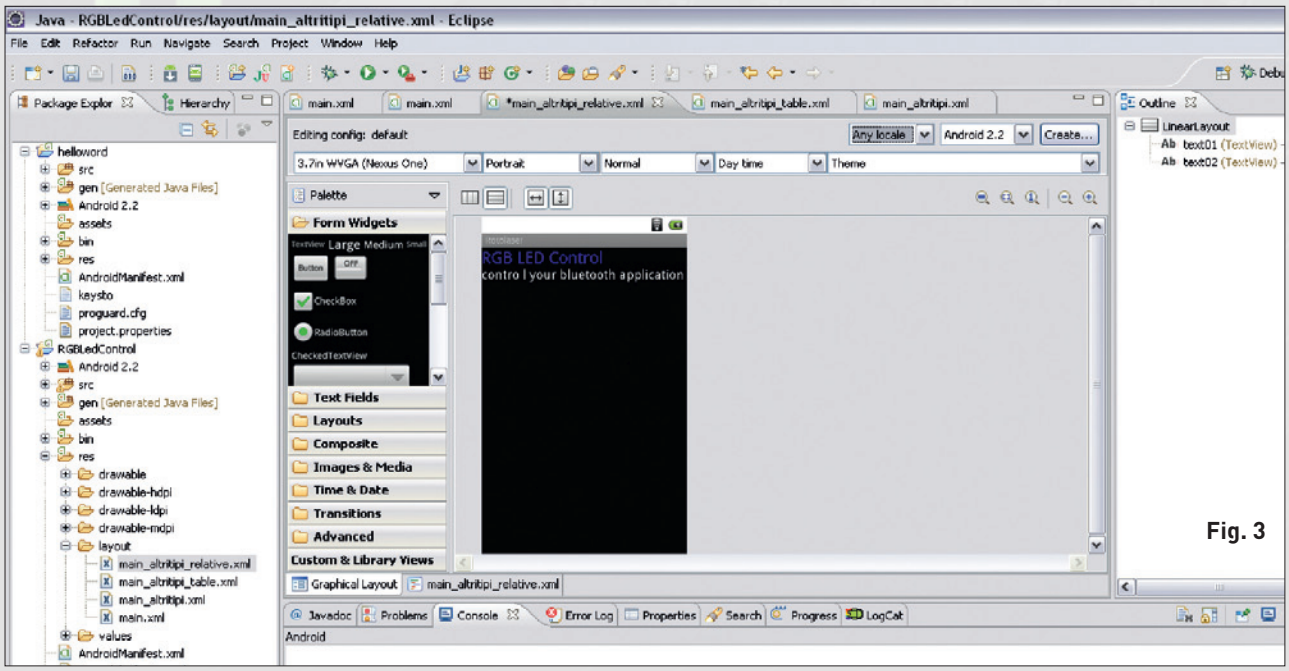


Fig. 3

Listato 3

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
                              android:layout_width="fill_parent"
                              android:layout_height="fill_parent"
                              android:orientation="vertical" >

    <TextView
        android:id="@+id/text01"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="34"
        android:textColor="#00F"
        android:text="RGB LED Control" />

    <TextView
        android:id="@+id/text02"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="20"
        android:textColor="#FFF"
        android:text="control your bluetooth application" />

</LinearLayout>
```

yout) , mostrato nel **Listato 2** nel quale i controlli vengono inseriti in sequenza, nello spazio a disposizione sullo schermo; uno sotto l'altro (nel caso in cui la sua proprietà *orientation* sia settata a *vertical*) oppure uno a fianco all'altro (nel caso in cui orientation sia settata a *horizontal*). Occorre prima fare una precisazione essenziale, senza la quale potremmo riscontrare alcuni problemi di incompatibilità con quanto scritto sopra: tutti i controlli, ma anche gli stessi *layout*, hanno sempre due proprietà fondamentali, *layout_width* e *layout_height* che possono assumere due valori, *wrap_content* e *fill_parent* (a partire dalle API Level 8 è stato rimpiazzato da *match_parent*). Useremo *fill_parent* se vogliamo estendere la dimensione del controllo fino al controllo "padre" (nel nostro caso il *LinearLayout* principale), mentre usereno *wrap_content* se vogliamo limitare la dimensione del controllo nei limiti del necessario. Questo vuol dire che se al nostro TextView del **Listato 2** (in cui il *LinearLayout* è impostato con orientazione verticale) volessimo aggiungere un altro controllo e visualizzarlo nella zona sottostante, dovremmo settare *layout_height* del primo TextView a *wrap_content*, altrimenti settandolo a *fill_parent* occuperebbe tutto lo shermo a disposizione ed il secondo TextView rimarrebbe coperto, proprio come se non esistesse.
Il file xml risultante dopo aver aggiunto il controllo è mostrato nel **Listato 3** ed il risultato grafico è mostrato in **Fig. 3**. Allo stesso modo, se il *LinearLayout* avesse orientazione orizzontale e volessimo inserire un altro controllo a fianco, questa volta sarà la proprietà *layout_width* che dovrà essere impostata a *wrap_content*. Da notare, inoltre, anche la proprietà id che deve essere diversa da un controllo all'altro specie quando dovremo utilizzare il controllo dal codice java. Il **Listato 3** è, alla fine, semplicemente lo stesso file *main.xml* che è già stato scritto al momento della creazione del progetto a cui abbiamo aggiunto un altro *TextView* e modificato alcune proprietà, come il colore blu (nel formato #RGB), il testo e la dimensione (*textSize*). Ora possiamo aggiungere i controlli per interagire effettivamente con lo shield RGB e modificare le componenti di colore del nostro dispositivo luminoso.
Visto che utilizzeremo il Bluetooth occorrerà un pulsante con cui inizializzare o terminare la comunicazione (useremo nello specifico un *ToggleButton*) e tre slide (in Android chiamate *SeekBar*) con cui cambiare le tre componenti di colore in modo indipendente.

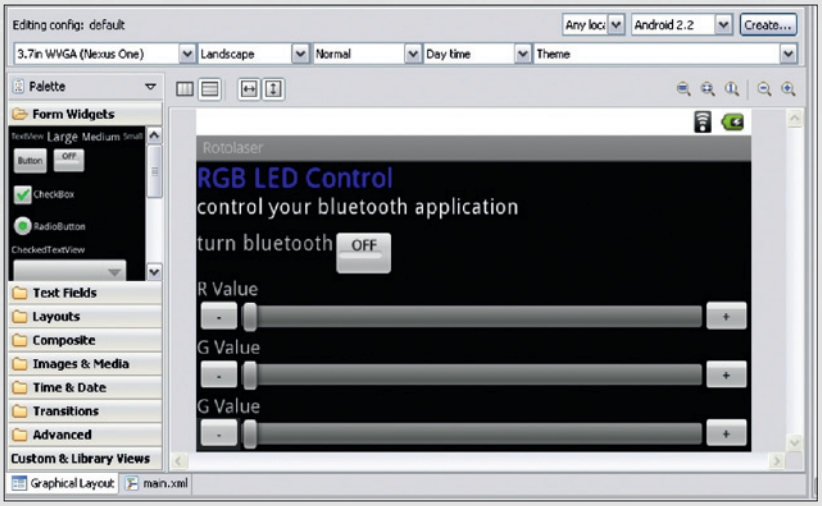


Fig. 4

Per impratichirci maggiormente con l'aspetto grafico, aggiungeremo ad entrambi i lati di ogni singola *SeekBar* un bottone (Button) con cui regolare più finemente la luminosità. Per fare questo, utilizzeremo lo stesso *LinearLayout* impostato verticalmente, ma questa volta conterrà al suo interno altri *LinearLayout* impostati orizzontalmente dove disporremo i due bottoni e la slide. È infatti possibile annidare più *layout* uno dentro l' altro ed anche di formati diversi, come vedremo. Nel **Listato 4** è visualizzato il file *main.xml* con l' inserimento della slide che cotrolla la componente del rosso (R). All'interno del layout principale abbiamo aggiunto un altro *TextView* (con *id/textViewR*) che utilizzeremo da codice per scrivere il valore di intensità della singola componente di colore. Da notare la proprietà *layout_weight* che stabilisce una proporzione tra i vari controlli contenuti all' interno di un *LinearLayout*: inserendo un valore più elevato per la slide al centro, questa si estenderà per una larghezza maggiore rispetto ai due bottoni laterali che sono impostati con valore molto più piccolo (1). Ovviamente le proprietà di ogni controllo sono molto numerose e potrete sbizzarrirvi ad aggiungerne e a modificarle a seconda dei vostri gusti grafici, senza dover ogni volta ricompilare il progetto, ma semplicemente passando alla modalità grafica (tab *Graphical Layout*) dopo aver scritto il file xml. Inoltre, un altro vantaggio offerto da Eclipse è

Listato 4

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
                    android:layout_width="fill_parent"
                    android:layout_height="fill_parent"
                    android:orientation="vertical" >

    <TextView
        android:id="@+id/text01"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="34"
        android:textColor="#00F"
        android:text="RGB LED Control" />

    <TextView
        android:id="@+id/text02"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="20"
        android:textColor="#FFF"
        android:text="control your bluetooth application" />

    <LinearLayout
         android:paddingTop="34dp"
         android:orientation="horizontal"
         android:layout_width="fill_parent"
         android:layout_height="wrap_content" >
            <TextView
                 android:id="@+id/text03"
                 android:layout_width="wrap_content"
                 android:layout_height="wrap_content"
                 android:text="turn bluetooth" />

            <ToggleButton
                 android:id="@+id/toggleButtonOnOff"
                 android:layout_width="wrap_content"
                 android:layout_height="wrap_content"
                 android:text="ToggleButton" />
    </LinearLayout>

    <TextView
        android:id="@+id/textViewR"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="25"
        android:text="R Value" />

    <LinearLayout
         android:orientation="horizontal"
         android:layout_width="fill_parent"
         android:layout_height="fill_parent"
         android:layout_weight="1">


        <Button
                  android:id="@+id/buttonRMinus"
                  android:layout_width="wrap_content"
                  android:layout_height="wrap_content"
                  android:layout_weight="1"
                  android:text="-" />

          <SeekBar
         android:id="@+id/seekbarR"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
                  android:max="20"
         android:layout_weight="25" />

          <Button
                  android:id="@+id/buttonRPlus"
                  android:layout_width="wrap_content"
                  android:layout_height="wrap_content"
                  android:layout_weight="1"
                  android:text="+" />
         </LinearLayout>

</LinearLayout>
```

rappresentato dalla possibilità di avere il completamento automatico della sintassi, così vi basterà scrivere il namespace "android:" ed avrete sul menu contestuale tutte le possibili proprietà che potrete utilizzare. Al **Listato 4** basterà ora aggiungere altri due *LinearLayout* per gestire le componenti Verde e Blu, ricordandosi di cambiare Id ai controlli, in maniera corrispondente. Otterremo un aspetto grafico come mostrato in **Fig. 4**.
Nella modalità grafica di Eclipse è anche possibile visualizzare lo schermo virtuale in modalità *landscape*, in modo tale da verificare la coerenza dell'interfaccia anche nel caso in cui il dispositivo reale venga ruotato di 90 gradi.
Alternativamente si può anche decidere che la nostra applicazione venga visualizzata a schermo, sempre e solo in una modalità, ad esempio landscape come in questo caso in cui avere una slide più larga ci permetterà una miglior regolazione. Per fare questo dovremo settare una proprietà nel file *AndroidManifest.xml* ed in particolare *screenOrientation* = "Landscape". Ora in qualsiasi posizione ruoteremo il nostro dispositivo Android, il *layout* si presenterà sempre con un solo orientamento.

Listato 5

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
                                android:layout_width="fill_parent"
                                android:layout_height="fill_parent"
                                android:stretchColumns="1">
            <TableRow>
                <TextView
                        android:id="@+id/textViewR"
                        android:layout_width="fill_parent"
                        android:layout_height="wrap_content"
                        android:textSize="18dp"
                        android:text="R Value" />
            </TableRow>

            <TableRow>
                  <Button
                          android:id="@+id/buttonRMinus"
                          android:layout_width="wrap_content"
                          android:layout_height="wrap_content"
                          android:layout_weight="1"
                          android:text="-" />
                   <SeekBar
                        android:id="@+id/seekbarR"
                        android:layout_width="wrap_content"
                        android:layout_height="wrap_content"
                        android:layout_weight="25" />
                    <Button
                          android:id="@+id/buttonRPlus"
                          android:layout_width="wrap_content"
                          android:layout_height="wrap_content"
                          android:layout_weight="1"
                          android:text="+" />
            </TableRow>

  </TableLayout>
```

Altri layout

Fin qui abbiamo analizzato il *Linear Layout* che è anche quello più utilizzato, ma in qualche altra applicazione potremo trarre maggiori vantaggi o trovarci semplicemente più comodi usando altri tipi di *layout*. Se ad esempio volessimo disporre oggetti in forma tabellare potremmo usare il *TableLayout* che facilita la composizione. Nulla vieta di utilizzare questo tipo di *Layout* anche per la nostra applicazione; nel **Listato 5** è mostrato il codice per inserire uno dei nostri slide utilizzando il *TableLayout* all' interno del *LinearLayout* principale.

Listato 6

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
android"
                                android:layout_width="fill_parent"
                                android:layout_height="fill_parent"
                                android:paddingBottom="30px"
                                android:padding="0px" >

        <Button
                android:id="@+id/buttonRMinus"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_weight="1"
                android:text="-" />

        <SeekBar
        android:id="@+id/seekbarR"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@+id/buttonRMinus"
        android:minWidth="200dp" />

         <Button
                android:id="@+id/buttonRPlus"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_toRightOf="@+id/seekbarR"
                android:text="+" />

</RelativeLayout>
```

Listato 7

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
                              android:layout_width="fill_parent"
                              android:layout_height="fill_parent" >
          <ImageView
            android:id="@+id/image01"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginLeft="40dp"
            android:src="@drawable/rgb" />

<LinearLayout
                    android:layout_width="fill_parent"
                    android:layout_height="fill_parent"
                    android:orientation="vertical" >

                    …

</LinearLayout>

</FrameLayout>
```

Il codice si commenta da solo. Un altro tipo di *layout* che possiamo utilizzare è il *RelativeLayout* in cui la posizione dei nostri controlli è relativa agli altri già esistenti.Utilizzando questo tipo di *layout* avremo a disposizione alcune proprietà particolari (come *layout_toRightOf, layout_toLeftOf, layout_below, layout_above*) e sarà necessario specificare l'id del controllo di riferimento. Nel **Listato 6** viene mostrato lo stesso esempio, ma utilizzando questo tipo di *layout*.
Il *RelativeLayout* ci permette di posizionare i nostri controlli in modo indipendente dalle dimensioni dello schermo che sono spesso molto diverse col variare del dispositivo finale. Per questo motivo è preferibile ad un altro tipo di *layout*, l'*AbsoluteLayout*, in cui le posizioni degli oggetti sono determinate, appunto, da coordinate assolute. L'*AbsoluteLayout* è sconsigliato dalla comunità Android e pertanto non lo analizzeremo in questa sede. Un altro tipo di *layout* è il *FrameLayout* in cui i controlli vengono posizionati tutti a partire dall'angolo superiore sinistro dello schermo in ordine di scrittura, ottenendo pertanto una sovrapposizione dei controlli. Questo può essere utilizzato anche nella nostra applicazione per inserire un'immagine di sfondo. Occorre inserire l'intero *LinearLayout* principale all' interno di un *FrameLayout* ed anteporre un nuovo controllo adatto per contenere immagini, l'*ImageView*. In questo controllo setteremo la proprietà src con il path relativo dell'immagine che avremo caricato nella cartella drawable (interna alla sottocartella *res*) e di seguito copieremo i controlli già utilizzati precedentemente, che ora verrrano sovrapposti all'immagine, ottenendo il risultato in **Fig. 5**. Da notare, anche in questo caso, come il nome del file immagine debba essere scritto nel file xml, senza estensione. Se poi la nostra applicazione avesse bisogno di molti più controlli di quelli visualizzabili su schermo, potremmo utilizzare un controllo particolare, chiamato *ScrollView* che ci permetterà di spostarci nella parte nascosta tramite una classica scrollbar. Occorrerà inserire i controlli, ma anche interi layout, all'interno della *ScrollView* come nel **Listato 8**.

Listato 8

```
<ScrollView
android:layout_width="fill_parent"   android:layout_height="wrap_content">

          <RelativeLayout>
              …
          </RelativeLayout>

</ScrollView>
```

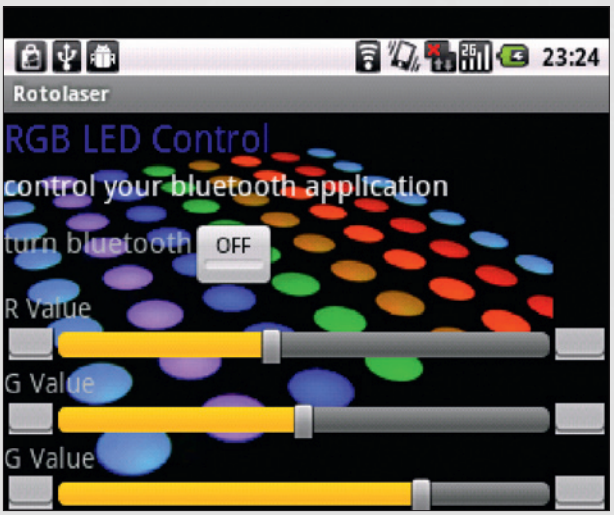


Fig. 5

Conclusioni

A questo punto, terminato l'aspetto grafico della nostra applicazione, ci possiamo preoccupare di scrivere il codice associato alle azioni che verranno eseguite sui nostri controlli, ovvero la gestione degli eventi, che analizzeremo nella prossima puntata. ■

Proseguiamo con la descrizione dell'applicazione per controllare, tramite smartphone, la luminosità di una striscia a LED RGB, utilizzando il protocollo Bluetooth e presentando il progetto di un apposito shield per Arduino. Quarta puntata.

di Andrea Chiappori

4

PROGRAMMIAMO CON ANDROID

Abbiamo visto, nella puntata precedente, come gestire l'interfaccia grafica su un dispositivo Android; lo abbiamo fatto usando come esempio l'inserimento nella nostra applicazione di tre barre scorrevoli per variare la tonalità di colore di una striscia a LED RGB, collegata in remoto tramite un dispositivo Bluetooth e Arduino. Ci eravamo lasciati con la presentazione del layout grafico senza preoccuparci della gestione degli eventi; ora è giunto il momento di analizzare il codice Android che ci permetterà di associare alle azioni dell'interfaccia grafica gli eventi che ci interessano, in modo da prendere il controllo del modulo Bluetooth integrato nel nostro smartphone ed inviare dati ad un altro modulo Bluetooth esterno. Analizzeremo anche il modulo necessario per ricevere dati dal nostro smartphone, per tradurli in formato seriale e per poi trasferirli ad Arduino. Lo schema di principio dell'applicazione è rappresentato in **Fig. 1**.

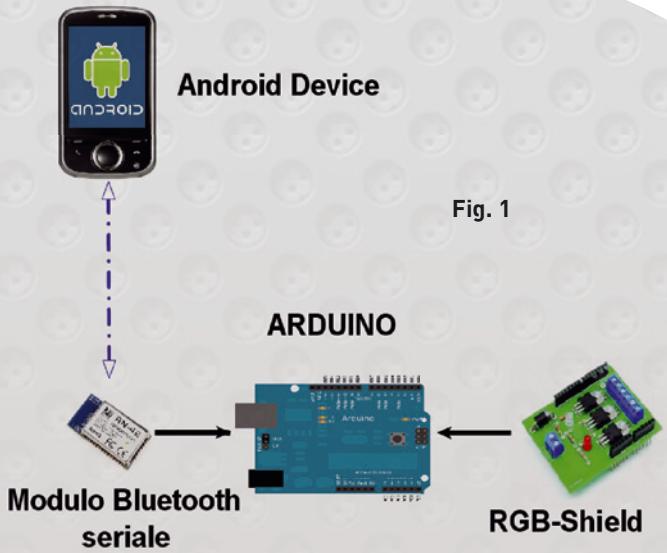


Fig. 1

CLASSI BLUETOOTH

1. **BluetoothAdapter**
   Rappresenta il modulo radio Bluetooth locale, ovvero quello del nostro smartphone. Da esso possiamo ottenere informazioni sullo stato (abilitato o meno), il nome, la classe, l'indirizzo e ancora altro.
2. **BluetoothDevice**
   Rappresenta il modulo Bluetooth remoto e di esso possiamo ottenere le stesse informazioni dell'adapter.
3. **BluetoothSocket**
   Rappresenta l'interfaccia tramite la quale avviene lo scambio di dati tra i due dispositivi Bluetooth.

CODICE ANDROID: GESTIONE BLUETOOTH

Incominciamo riprendendo il nostro progetto su Eclipse ed aggiungendo le funzionalità che ci permetteranno di gestire il modulo Bluetooth del nostro dispositivo. Tutto ciò ci viene permesso con facilità dalle API Android che abbiamo a disposizione dopo aver installato il plugin **Adt** (Android Development Toolkit) visto nella prima puntata, senza ricorrere all'uso di librerie esterne. Basterà semplicemente aggiungere i seguenti import nel codice principale, prima di dichiarare la classe:

```
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
```

Saremo, pertanto, in grado di utilizzare le classi Bluetooth che utilizzeremo nel nostro progetto e che sono brevemente descritte nel riquadro CLASSI BLUETOOTH che trovate in queste pagine. Prima di analizzare nel dettaglio il codice è opportuno ricordarsi di aggiungere nel file *AndroidManifest.xml* del nostro progetto l'autorizzazione ad usare il modulo Bluetooth dello smartphone; potete fare ciò attraverso le seguenti righe di codice:

```
<manifest>
…
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
...
</manifest>
```

Queste informazioni, oltre ad essere usate nei filtri interni di Google Play (il market-store di Android) servono ad avvertire l'utilizzatore finale, al momento dello scaricamento dell' applicazione, che questa utilizzerà il modulo Bluetooth.
In una comunicazione Bluetooth tra due dispositivi, uno si comporterà come *server* e l'altro come *client*. Il server viene attivato per primo e mette a disposizione un "server socket" sul quale "viaggeranno" i dati in entrambe le direzioni, ma in modalità half-duplex, ovvero non simultaneamente. Il client, per comunicare con il server, dovrà prima connettersi al "server socket" (tramite funzione *connect()*) utilizzando il "MAC address" del dispositivo che funziona da server.
Nel nostro progetto utilizzeremo lo smartphone come client ed il modulo Bluetooth esterno come server. Il MAC address è un identificatore univoco di sei byte assegnato alle interfacce di rete; conoscendolo, siamo sicuri che il client comunicherà esclusivamente col server scelto.
Come fare a conoscere questo indirizzo? Solitamente il client effettua una scansione sul Bluetooth per verificare la presenza di altri dispositivi remoti attivi, da cui ricava i relativi MAC address in modo che l'utente possa scegliere a quale di questi server connettersi. Una volta scelto un MAC address, otterremo le informazioni del dispositivo remoto con le seguenti righe di codice:

```
mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
BluetoothDevice device = mBluetoothAdapter.
getRemoteDevice(serverAddress);
```

Da notare come *mBluetoothAdapter* rappresenti il dispositivo locale ottenuto tramite il metodo *getDefaultAdapter()* e grazie al metodo *getRemoteDevice()* a cui viene passato il MAC address del dispositivo remoto, otteniamo un *BluetoothDevice* dal quale avremo informazioni sul suo nome, indirizzo e classe. Infine, sempre dal dispositivo remoto possiamo ottenere il server socket (*btSocket*) come nella seguente riga di codice:

```
btSocket = device.createRfcommSocketToServiceRecord(MY_UUID);
```

e per connetterci al server useremo la riga:

```
btSocket.connect();
```

Per ora ci occuperemo di inviare dati in una sola

direzione, ovvero dal nostro smartphone ad Arduino (al quale sono collegati sia la barra a LED, sia il modulo Bluetooth). Questo vuol dire che una volta connessi al socket Bluetooth del server, lo useremo solo in scrittura, inviando i dati relativi alle componenti di colore scelte dall'interfaccia, utilizzando le linee di codice:

```
outStream = btSocket.getOutputStream();
outStream.write(data);
```

Per adesso e per semplicità, utilizzeremo un MAC address cablato nel codice e uguale a quello del nostro modulo Bluetooth. Per ottenere tale valore possiamo leggerlo sull'etichetta del dispositivo dove solitamente è scritto, oppure ricavarlo interrogando quest'ultimo con comandi seriali, come spiegato in seguito. Più avanti mostreremo come eseguire una scansione di tutti i server Bluetooth attivi ed ottenere il MAC address direttamente dalla nostra applicazione per rendere il codice più usabile e modulare.
Ricapitolando, alla pressione del nostro pulsante "OnOff" dovremo inizializzare il modulo Bluetooth, verificando prima che il nostro smartphone lo supporti e successivamente che sia abilitato.
In caso di errore visualizzeremo un messaggio di errore in una finestra di dialogo a scomparsa che Android chiama *MessageToast* (**Listato 1**).
A connessione avvenuta, ad ogni tocco sulla barra di scorrimento verrà inviata una stringa con un messaggio del tipo: "R123\n" verso il dispositivo Bluetooth remoto, dove con R viene indicata la componente di colore rosso (G nel caso del verde e B nel caso del blu), con "123" il valore della componente di colore che può variare da 0 a 255 e con "\n" il ritorno a capo per determinare la fine del messaggio.
La scelta della sintassi del messaggio contenuto nella stringa è personalizzabile a piacere. Potremmo inviare stringhe di qualsiasi tipo: l'importante è che vengano coerentemente parserizzate ed interpretate dal modulo remoto che le riceverà; nel nostro caso sarà Arduino ad occuparsi di ciò (in seguito vedremo come).
In ogni caso, l'invio di questo messaggio è reso possibile proprio accedendo in scrittura al server socket visto prima, come viene fatto nella funzione del **Listato 2**.
Come si vede nei listati 1 e 2, a corredo del nostro codice e per semplificarne il debug abbiamo aggiunto nei punti strategici qualche riga di log.

Listato 1

```
int initRet = BtCore.initBluetooth();
Log.v(TAG, "initBluetooth: "  +  initRet);

if (initRet == -1)
          {
          Toast.makeText(BtclientActivity.this, "Bluetooth
          is not available.", Toast.LENGTH_LONG).show();
          Log.e(TAG, "error -1");
          finish();
}
if (initRet == -2)
{
          Toast.makeText(BtclientActivity.this, "Please
          enable your BT and re-run this program.", Toast.
          LENGTH_LONG).show();
          Log.e(TAG, "error -2");
          finish();
}

int crtRet = BtCore.createSocketBluetooth();
Log.v(TAG, "createSocketBluetooth: "  +  crtRet);

retCon = BtCore.connectBluetooth();
Log.v(TAG, "connectBluetooth:" + retCon);

if (retCon != 0)
{
          Toast toastStart = Toast.makeText(BtclientActivity.
          this, "no module bluetooth found!", Toast.LENGTH_LONG);
          toastStart.show();
          Log.v(TAG, "exit");

          _toggleOnOff.setChecked(false);

          return;
}
Log.v(TAG, "On");
```

Da notare come l'API Log può avere diversi livelli a seconda della sua rilevanza: *Log.e()* è utilizzata per identificare un messaggio di errore, *Log.v()* un semplice messaggio informativo e *Log.w()* un messaggio di attenzione (warning). Questa diversità viene tradotta in un diverso colore con cui LogCat (il monitor Android) mostra questi messaggi. La **Fig. 2** mostra un esempio d'uso di Log().
Il TAG, invece, è sempre una stringa di testo e

Listato 2

```
public static void sendMessageBluetooth(String message)
{
          if (outStream == null)
          {
                    Log.e(TAG, "stream is Null");
                    return;
          }
          byte[] msgBuffer = message.getBytes();
          try
          {
                    outStream.write(msgBuffer);
                    Log.v(TAG, "btSended:" + message);
          }
          catch (IOException e)
          {
                    Log.e(TAG, "seekbarLeft: Exception during write.", e);
          }
}
```

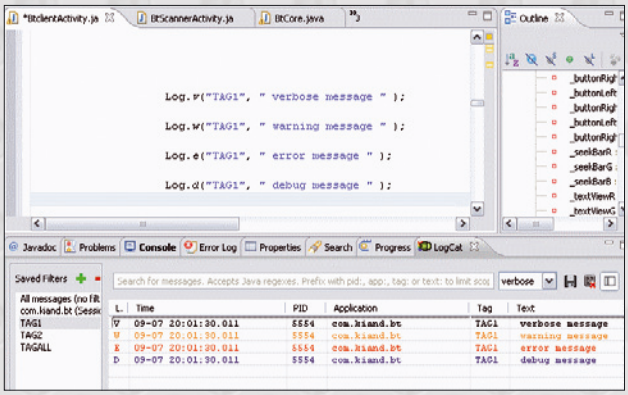


Fig. 2

serve per differenziare i messaggi di log in modo da poterli filtrare ed analizzarli senza mischiarli con quelli di sistema.
Inoltre LogCat è attivo anche senza lanciare l'applicazione in modalità di debug dell'applicazione, ma semplicemente collegando il cavo USB ed abilitando la modalità debug nel nostro dispositivo Android.
Tutte le funzioni per la gestione del protocollo Bluetooth sono state inserite in una classe nuova (BtCore), contenuta nel file *BtCore.java* in modo da poter essere riutilizzata in seguito nelle nostre future applicazioni che richiederanno l'uso di un modulo Bluetooth. Occorre, però, ricordarsi che i file java dello stesso progetto (all'interno della cartella *src*) dovranno avere come intestazione lo stesso package, che nel nostro caso è "package com.kiand.bt".

CODICE ANDROID: EVENTI

Ora vediamo come associare queste azioni all'interfaccia grafica della nostra applicazione: ad esempio al pulsante *_toggleOnOff* che useremo per creare il socket Bluetooth. Prima di tutto definiamo un oggetto (detto anche *widget*) *ToggleButton* (ricordandoci sempre di definire il corrispettivo import *android.widget.ToggleButton*); nella funzione *oncreate*() dell'Activity principale, associeremo tale oggetto alla risorsa precedentemente generata nel file *main.xml*, tramite la funzione *findViewById()*:

_toggleOnOff =(ToggleButton)findViewById(R.id.toggleButtonOnOff);

A questo punto possiamo modificare le proprietà del pulsante *_toggleOnOff* a nostro piacimento, modificando ad esempio la scritta (tramite metodo *setText()*) o il colore (tramite il metodo *setBackgroundColor()*) nel modo che appare nel listato; possiamo quindi associare la funzione (callback) che verrà chiamata quando clicchemo sul pulsante. In realtà, non verrà passata esattamente la funzione, ma una classe di interfaccia *view.OnClickListener* che contiene già alcuni metodi tra cui la *onClick()*, che potremo riempire a nostro piacimento. Solitamente questa classe viene creata direttamente (con l'operatore *new*) nel parametro di passaggio, ottenendo un codice come quello del **Listato 3**.
Nel caso del *toggleButton*, che ammette due stati, sul relativo metodo *onClick()* andremo a verificare in quale dei due stati si trovi (accesso o spento) e ci comporteremo di conseguenza andando a creare il socket di comunicazione con le funzioni Bluetooth create ed analizzate precedentemente. Se cliccheremo una seconda volta sul pulsante, chiuderemo il socket Bluetooth.
Il discorso è più semplice per i pulsanti normali che nella puntata precedente avevamo disposto ai lati delle singole progress-bar per regolare finemente la luminosità.
La funzione per settare la callback sul click è sempre la *setOnClickListener()* e nella *onClick()* incrementeremo (per il pulsante +) il valore aggiornando la barra di scorrimento tramite metodo *setProgress()* e cambieremo opportunamente il valore della stringa di testo *_textViewG*, tramite il metodo *setText()*. Naturalmente, alla fine invieremo il valore raggiunto verso Arduino usando la nostra funzione *sendMessageBluetooth()* che scriverà il messaggio (ad esempio la stringa "G234\n") sul socket di comunicazione.
Per l'evento sullo spostamento delle barre di scorrimento tramite touch, useremo un metodo

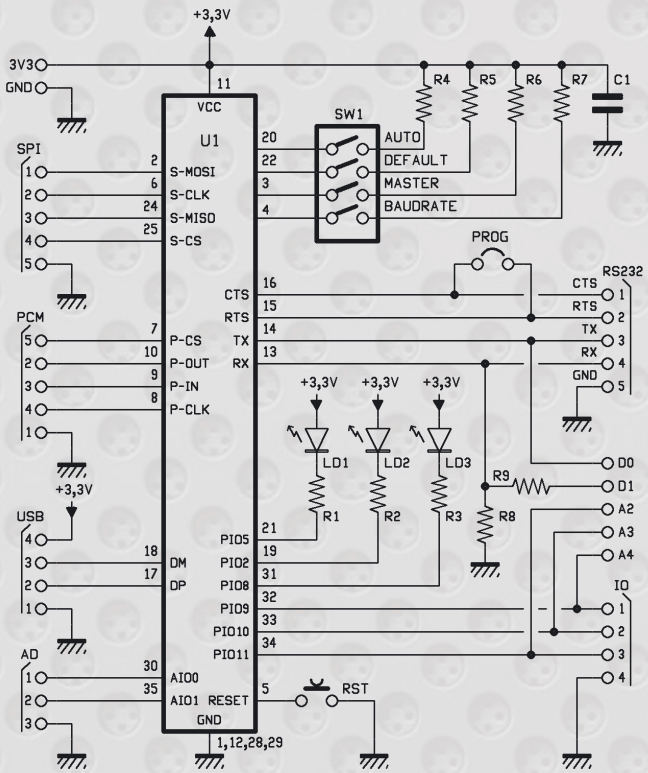
Listato 3

```
_toggleOnOff = (ToggleButton) findViewById(R.id.toggleButtonOnOff);
_toggleOnOff.setBackgroundColor(Color.GREEN);
_toggleOnOff.setText("ON");
_toggleOnOff.setOnClickListener(new View.OnClickListener()
{
        public void onClick(View v)
        {

                if (_toggleOnOff.isChecked())
                {
                        // ...
                        // attivazione Bluetooth
                        //...
                }
                else
                {
                        // ...
                        // chiusura Bluetooth
                        // ...
                }
        }
}
```

diverso: mentre per i pulsanti abbiamo creato tutto all'interno del metodo *onCreate()* dell'activity, che viene eseguita solo la prima volta, per la progress bar utilizzeremo il metodo *onProgressChanged()* che sarà proprio dell'activity stessa. In ogni caso nella *onCreate()* sarà necessario, prima di tutto, associare l' oggetto seekBar alla risorsa corrispondente e settare la callback a sè stessa, come visibile nel **Listato 4**. In questo modo, ogni volta che toccheremo una delle tre barre di scorrimento ci troveremo nella *onProgressChanged()* che è dichiarata direttamente nell'activity (notare che è stato necessario usare *implement SeekBar.OnSeekBarChangeListener* per poterla utilizzare all'interno della classe stessa). In particolare, la callback *onProgressChanged()* ha tre parametri, visibili sempre nel **Listato 4**, che al momento del touch saranno riempiti con i valori appropriati in modo da poter rilevare quale delle tre barre di scorrimento (seekBar) è stata mossa e con quale valore (progress).
Il primo valore verrà confrontato con ognuna delle risorse definite nel file *main.xml* (grazie al metodo *getId()*) e il secondo parametro verrà inviato come stringa verso Android. Infine, il terzo parametro avrà valore True o False a seconda che il movimento sia stato generato o meno dall'utente; tuttavia nel nostro caso non verrà preso in considerazione.
A questo punto la nostra applicazione Android ha tutto l'essenziale per controllare una piattaforma hardware esterna, che sarà ovviamente dotata di un modulo Bluetooth. Spetterà al cuore

Listato 4

```
@Override
public void onCreate(Bundle savedInstanceState)
{
// ...
          _seekBarR = (SeekBar)findViewById(R.id.seekbarR);
          _seekBarR.setOnSeekBarChangeListener(this);
// ...
}

public void onProgressChanged(SeekBar seekBar, int
progress, boolean fromTouch)
{
          int id = seekBar.getId();

          if (id == R.id.seekbarR)
          {
            _RValue = progress;
            _textViewR.setText("R:" + progress);
            BtCore.sendMessageBluetooth("R" + progress +"\n");
          }
//…
}
```



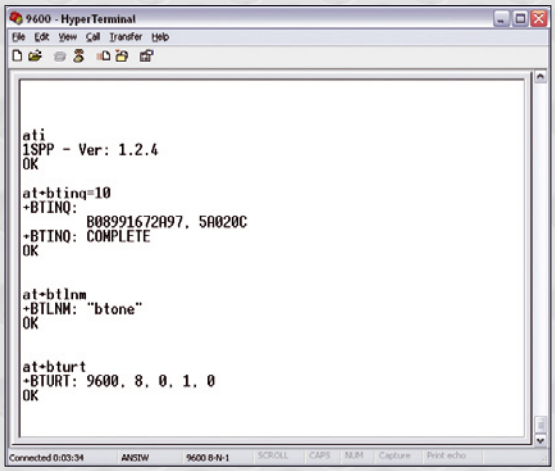
Schema elettrico dello shield Bluetooth per Arduino descritto nell'articolo.

della scheda remota (nel nostro caso Arduino), gestire i messaggi ricevuti in formato seriale ed interpretarli per svolgere le funzionalità che ci interessano (nel nostro caso, il controllo della barra a LED).
Potremmo passare alla descrizione del codice su Arduino, ma andiamo per gradi e occupiamoci prima del modulo Bluetooth esterno che rappresenta il tramite per la comunicazione tra Android ed Arduino.

MODULO BLUETOOTH

Esistono tanti modelli di moduli Bluetooth che si basano su chipset differenti e che, pertanto, richiedono un diverso tipo di pilotaggio; si differenziano per numero di pin e rispettive funzionalità, ma tutti possono comportarsi sia da server che da client e supportano il profilo SPP (Serial Port Profile) ovvero la capacità di emulare una seriale RS232 convertendo i segnali radio in entrambe le direzioni. Inoltre, proprio come una comune porta seriale, possono essere collegati direttamente ad un microcontrollore tramite i soli due fili TX ed RX e sono in grado di gestire diverse velocità di comunicazione.
Come fare, allora, per configurarli, se sono collegati con soli due fili? Ebbene, esistono due

Fig. 3



diverse modalità di funzionamento: una di comando e una dati.
Nella modalità comando i segnali della seriale vengono usati per impartire istruzioni al nostro modulo, come ad esempio per settare la configurazione (server/client) o per richiedere informazioni sulla versione, conoscere il MAC (di cui abbiamo già parlato), il nome (e tante altre caratteristiche) alle quali il modulo risponderà con stringhe di conferma o di errore.
Nel nostro progetto abbiamo utilizzato uno shield con a bordo un modulo Bluetooth di ultima generazione (RN-42), i cui comandi e settaggi sono già stati ampiamente documentati sul numero 169 della rivista, ma volendo utilizzare anche moduli Bluetooth più datati che si basano su sistema "BlueSMiRF" si può fare riferimento ai comandi AT descritti nella documentazione del produttore e ad un piccolo esempio di comunicazione tra modulo e seriale del PC (vedere **Fig. 3**).
Normalmente, dando alimentazione al modulo, questo entra direttamente nella modalità dati; in questa, i pin della seriale vengono utilizzati esclusivamente per scambiarsi messaggi.
Per entrare nella modalità comando, invece, viene usata una particolare sequenza di caratteri, detta sequenza di escape, che nel nostro caso è "$$$". Quindi inviando sul piedino RX del modulo Bluetooth questa sequenza, il modulo risponde con una sua stringa di conferma e rimane in attesa di altri comandi. Per ritornare alla modalità dati, esistono altre sequenze di escape che variano da modulo a modulo.
Possiamo inviare questi comandi al modulo Bluetooth utilizzando la seriale di Arduino; nel nostro progetto faremo proprio così.
Se volete fare un po' di pratica con il modulo Bluetooth, ad esempio per imparare come configurare la seriale e fare altre impostazioni, vi consigliamo di collegarlo alla seriale del PC tramite uno di quei convertitori di livello seriale (per adattare i 12 V del PC ai 3,3 V del modulo) ed usare HyperTerminal (o qualsiasi altro terminale seriale su PC); potrete così fare tutte le prove che desiderate.
Sapere già come gestire il modulo Bluetooth vi eviterebbe anche di compilare ogni volta il codice su Arduino, soprattutto in fase di test.

LO SHIELD BLUETOOTH RN-42

Per realizzare l'esercitazione descritta in questa puntata del corso abbiamo preparato uno shield per Arduino basato sul modulo RN-42: si tratta di un circuito comunque utilizzabile in tutte le applicazioni in cui serva instaurare una connessione su Bluetooth. Di seguito ne diamo una breve descrizione, partendo dai connettori di cui dispone, che servono ad interfacciarlo con il mondo esterno; escludiamo quelli di connessione con Arduino. Il primo è quello siglato RS232, le cui connessioni sono le seguenti:

- 1 = CTS;
- 2 = RTS;
- 3 = TX;
- 4 = RX;
- 5 = GND.

Tale connettore è l'interfaccia seriale grazie alla quale avviene la comunicazione di dati e comandi. Si accede alla fase di comando inviando la sequenza di escape "$$$<cr>" sul pin RX del modulo Bluetooth, tramite la seriale locale del PC o direttamente da Arduino. Per uscire da questa fase ed entrare nella fase dati occorre inviare la sequenza "---".
Durante la fase dati ed una volta instaurata una connessione Bluetooth, ciò che viene inviato sul pin RX verrà diretto, via radio, al dispositivo remoto, mentre sul pin TX ritroveremo ciò che viene ricevuto dal dispositivo remoto.
Solitamente si usano solo questi tre pin (TX, RX e GND), mentre CTS ed RTS vengono impiegati per il controllo di flusso, ma rimangono comunque disponibili.
Nel caso si voglia utilizzare anche CTS ed RTS occorre scollegare il jumper PROG. Il pin CTS può essere anche usato per "svegliare" il modulo Bluetooth dalla fase di sleep (in cui consuma meno di 2 mA) mediante il passaggio da stato logico "0" a stato logico "1".
Il secondo connettore dello shield è quello sigla-

MASCHERA I/O								VALORE OUTPUT							
x	x	x	x	IO 11	IO 10	IO 9	IO 8	x	x	x	x	IO 11	IO 10	IO 9	IO 8
0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0

Fig. 4

to IO e raggruppa tre degli I/O disponibili nel modulo RN-42; la sua piedinatura è la seguente:
• 1 = GPIO9;
• 2 = GPIO10;
• 3 = GPIO11;
• 4 = GND.

Per leggere o pilotare questi I/O è necessario entrare nella modalità comandi (da seriale locale o direttamente da Arduino). Per controllare tali linee di I/O bisogna fare riferimento al comando **S*,<*word*>** in cui *word* è una stringa esadecimale di 16 bit i cui 8 bit più significativi rappresentano la maschera per determinare i pin di output e di input, mentre gli 8 bit meno significativi corrispondono al valore (1 per portare il pin a livello alto e zero per portarlo a livello basso). La struttura è quella mostrata nella **Fig. 4**.
Per esempio:
S*,0202 abilita GPIO9 come output e lo mette al valore "1"
S*,0E00 abilita mette tutti i GPIO dello strip a "0"

Il comando GK restituisce una stringa esadecimale col valore dei pin allo stato attuale; tali pin sono collegati anche ai pin di Arduino. In particolare:
• GPIO9 è collegato ad A4;
• GPIO10 è collegato ad A3;
• GPIO11 è collegato ad A2.

E adesso passiamo al connettore AD, i cui contatti sono solo ingressi analogici e possono essere utilizzati dal modulo Bluetooth per acquisire alcune informazioni, come ad esempio il livello di tensione sulla batteria. La piedinatura di questo connettore è:
• AIO0;
• AIO1;
• GND.

In questa puntata del corso il connettore IO non viene usato.
Passiamo adesso al connettore SPI, la cui piedinatura è:
• 1 = S-MOSI;
• 2 = S- SCLK;
• 3 = S-MISO;
• 4 = S-CS;
• 5 = GND.

Il connettore veicola un bus SPI e tramite esso si può scrivere nella Flash interna al modulo RN-42 in modo da aggiornare o sostituire il firmware; tale connessione non è comunque indispensabile.
Nello shield abbiamo anche un connettore siglato PCM, la cui piedinatura è la seguente:
• 1 = GND;
• 2 = P-OUT;
• 3 = P-IN;
• 4 = P-CLK;
• 5 = P-CS.

L'interfaccia PCM offre la possibilità di inviare via radio, ad un altro dispositivo Bluetooth, uno stream audio ad una velocità di trasferimento di 64 kbps. Per i nostri scopi questa possibilità non viene sfruttata.
L'ultimo connettore che vediamo è quello USB, che evidentemente veicola una connessione USB; la sua piedinatura è:
• 1 = GND;
• 2 = DP;
• 3 = DM;
• 4 = Vcc.

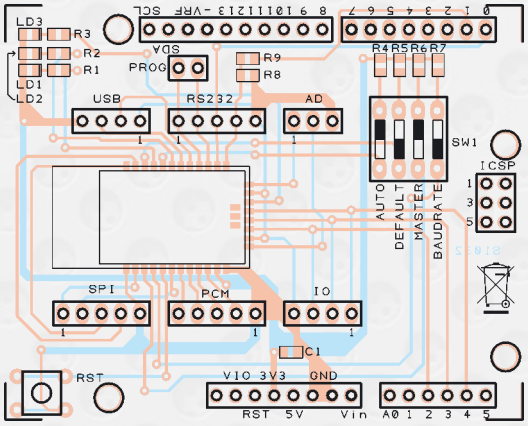
Collegando questo connettore al PC tramite un cavo USB è possibile utilizzare la modalità "HCI over USB" del modulo Bluetooth ed ottenere velocità maggiori (fino a 3 Mbps). Tuttavia, questo è possibile utilizzando solo la versione USB del modulo RN-42 (chiamata appunto RN-42-USB) o aggiornando opportunamente il firmware tramite il bus SPI.
Bene, analizzati i connettori vediamo gli altri elementi previsti nello shield, partendo dal dip-switch SW1, che può collegare alcuni pin (GPIO3, GPIO4, GPIO6, GPIO7) del modulo Bluetooth a VCC (3,3V) tramite quattro resistenze; questi pin servono per impostare alcune modalità di funzionamento del modulo RN-42 e sono normalmente collegati a massa tramite resistenze di pull-down integrate nel modulo. Spostando il rispettivo dip verso la scritta "ON" viene collegato il pin del modulo a Vcc e quindi a livello logico 1, abilitando la relativa funzionalità. Tutti questi pin vengono interpretati al momento dell'avvio, quindi cambiando lo stato dei dip durante il funzionamento dello shield

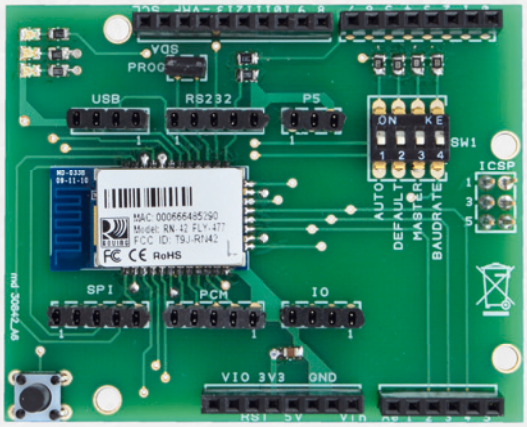
[Shield Bluetooth, piano di **MONTAGGIO**]

Elenco Componenti:

R1: 220 ohm (0805)
R2: 220 ohm (0805)
R3: 220 ohm (0805)
R4: 1 kohm (0805)
R5: 1 kohm (0805)
R6: 1 kohm (0805)
R7: 1 kohm (0805)
R8: 20 kohm (0805)
R9: 10 kohm (0805)
C1: 100 nF ceramico (0805)
U1: RN-42
RST: Microswitch
LD1: LED rosso (0805)
LD2: LED verde (0805)
LD3: LED giallo (0805)
SW1: Dip-Switch 4 vie

Varie:
- Strip M/F 3 vie (2 pz.)
- Strip M/F 6 vie
- Strip M/F 8 vie (2 pz.)
- Strip M/F 10 vie
- Strip Maschio 2 vie
- Strip Femmina 3 vie
- Strip Femmina 4 vie (2 pz.)
- Strip Femmina 3 vie (3 pz.)
- Jumper
- Circuito stampato





non si otterranno gli effetti desiderati.
Le modalità si possono anche configurare tramite software, entrando nella fase comando tramite la sequenza di escape "$$$".
Di seguito descriviamo i pin che fanno capo al dip-switch.

• **AUTO**. È collegato al pin GPIO3 del modulo Bluetooth. Se abilitato, permette l'accoppiamento automatico con un altro dispositivo remoto Blueto-oth, senza pertanto dover digitare il codice segreto (detto "Passkey"). Se però il dispositivo remoto è settato per richiedere l'autenticazione, verrà comunque richiesto l'inserimento della Passkey per instaurare un collegamento sicuro. La Passkey predefinita è 1234, ma può essere modificata, entrando in modalità comando, tramite sequenza di escape "$$$".

• **DEFAULT**. È collegato al pin GPIO4 del modulo Bluetooth. Permette di ripristinare tutti i settaggi interni al modulo ai valori iniziali di "fabbrica". Può essere utile quando vengono impartiti comandi che portano il modulo in conflitto. Per effettuare questo reset , occorre portare verso ON il relativo dip, prima di alimentare lo shield; una volta alimentato occorre riportarlo allo stato "0", poi allo stato "1" , di nuovo allo stato "0" e ancora ad "1". Ad ogni cambiamento di stato deve trascorrere un secondo circa.

• **MASTER**. È collegato al pin GPIO6 del modulo Bluetooth. Per impostazione predefinita, il modulo Bluetooth si comporta come un dispositivo slave e quindi può venire rilevato da altri dispositivi Bluetooth e connettersi ad essi, ma non è in grado di iniziare per primo una connessione. Portando a ON il rispettivo dip, il modulo lavorerà in modalità Master. In questa fase non potrà essere rilevato da altri dispositivi, ma sarà esso ad effettuare una connessione verso un particolare dispositivo remoto che avremo memorizzato in precedenza nella Flash del modulo RN-42, tramite il comando **SR,<*address*>** (in cui <*address*> è l'indirizzo MAC del dispositivo remoto scelto). Nel caso in cui non ci fosse alcun indirizzo memorizzato, l'RN-42 effettuerà una scansione e si connetterà al primo dispositivo remoto trovato. Sono inoltre possibili altre modalità Master che variano dal modo più o meno

particolare di comportarsi; queste si possono settare solo da software tramite il comando **SM,<x>**, in cui *x* rappresenta la modalità e varia da 0 a 5. In particolare SM,0 farà ritornare il modulo in modalità slave.

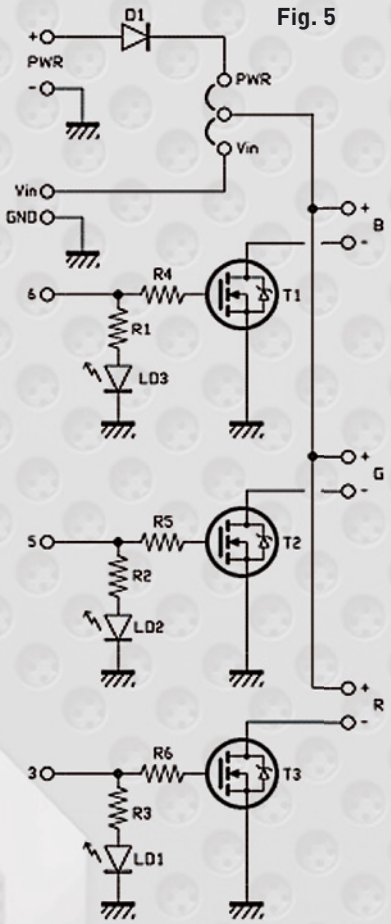
• **BAUDRATE**. Questo dip è collegato al pin GPIO7 del modulo Bluetooth. Abilitandolo, viene forzata la velocità di comunicazione per dati e comandi a 9.600 bps. Altrimenti verrà utilizzata la velocità desiderata che avremo precedentemente scritto in Flash tramite il comando **SU, <*rate*>** (esempio: SU,57.6 per selezionare 57.600 baud). Per impostazione predefinita, questo valore è pari a 115.200 baud. Tramite software è possibile impostare anche velocità di trasferimento non standard.

Notate che disponendo di due shield è possibile metterli in comunicazione punto-punto tra di loro per permettere comunicazione tra due PC (o altri dispositivi) evitando l'uso di un cavo seriale. In questo caso uno dei due dovrà comportarsi come master (spostando il relativo dipswitch) e l'altro rimarrà slave. Su entrambi dovrà essere settato il dipswitch AUTO in modo da non richiedere codice di autenticazione.

Bene, passiamo adesso al LED di stato LD1 (rosso) collegato al pin GPIO5 del modulo Bluetooth: all'avvio, nella modalità slave, lampeggia ad una frequenza di circa 1 Hz per indicare che è in attesa di essere rilevato e di ricevere la connessione. Quando si entra in modalità comando lampeggia a frequenza maggiore (circa 10 Hz). Una volta effettuata la connessione con un dispositivo remoto, rimane accesso.

Il secondo LED presente nel circuito (LD2, verde) collegato al pin GPIO2 del modulo Bluetooth, rimane sempre acceso e si spegne solamente durante la fase di connessione. Il terzo diodo luminoso (LD3, di colore giallo) è invece connesso al pin GPIO8 del modulo Bluetooth ed indica, pulsando, il passaggio di dati sia in una direzione che nell'altra, durante la fase di connessione.

Fig. 5



CODICE ARDUINO

A questo punto siamo pronti per analizzare il codice per Arduino, che riceverà i dati Bluetooth già convertiti in seriale dal modulo esterno e li tradurrà in azioni, ovvero cambierà tonalità alla nostra barra a LED. Questi dati Bluetooth altro non sono che i messaggi creati in precedenza (esempio "R200\n" per portare la componente del rosso ad un valore 200) e che arriveranno ad Arduino sul piedino RX della seriale. Arduino piloterà la barra a LED tramite uno shield RGB (già presentato nel fascicolo n° 159) il cui schema elettrico è visualizzato nella **Fig. 5**; lo shield viene alimentato a 12V ed utilizza tre MOSFET di potenza pilotati dai piedini analogici 3, 5 e 6 di Arduino.

Nella fase di setup del codice Arduino, inizializzeremo i piedini 3,5 e 6 come output e configureremo la seriale con le stesse proprietà e velocità del modulo Bluetooth al quale Arduino è collegato. Il modulo in questione ha una velocità di comunicazione predefinita di 115.200 bps, ma è possibile configurarlo con altri valori. Addirittura può funzionare a velocità non consentite da una normale seriale RS232 da PC, come ad esempio 1 Mbps. Noi lo abbiamo configurato a 9.600 bps.

È bene ricordare che utilizzando uno di questi moduli Bluetooth esterni, andremo ad occupare interamente la seriale di Arduino e ci mancherà un possibile feedback che risulta utile in fase di test e di debug per verificare lo stato di funzionamento del nostro codice.

In questo caso possiamo sempre ricorrere ad una libreria seriale esterna (*SoftwareSerial*) che ci permette emulare la porta seriale su due piedini qualsiasi di Arduino e perciò di tracciare tracciare e risolvere eventuali problemi.

Infine, sempre nella fase di setup, faremo entrare il modulo Bluetooth nella modalita' comando,

Listato 5

```
void setup()
{
  pinMode(RED_LED,   OUTPUT);
  pinMode(BLUE_LED,  OUTPUT);
  pinMode(GREEN_LED, OUTPUT);
  pinMode(BOARD_LED, OUTPUT);

  digitalWrite(GREEN_LED, HIGH);
  digitalWrite(BLUE_LED, HIGH);
  digitalWrite(RED_LED, HIGH);

  Serial.begin(115200);
  mySerialDbg.begin(9600);
  delay(500);
  mySerialDbg.println("Test RGB COntroller");

  delay(1000);

 // enter in command mode
  Serial.println("$$$");
  delay(1000);

  // force to slave mode
  Serial.println("SM,0");
  delay(3000);

  //enter in data mode
  Serial.println("---");
  delay(2000);

  lampLed(2);

  digitalWrite(GREEN_LED, LOW);
  digitalWrite(BLUE_LED, LOW);
  digitalWrite(RED_LED, LOW);
}
```

tramite l'invio della sequenza di escape su seriale e lo forzeremo in modalità slave, per poi farlo tornare alla fase di scambio dati. Per tutta la fase di setup abbiamo mantenuto i LED di tutti e tre i colori accessi (quindi la barra a LED risulterà di colore bianco); al termine, prima di passare al loop infinito, spegneremo tutti i LED (tramite funzione *digitalWrite()*) in modo da segnalare l'avvenuto passaggio nella fase dati. Nel **Listato 5** è possibile visualizzare questa parte di codice. Ora saremo in grado di controllare la luminosità della barra tramite la nostra applicazione Android; vediamo come. Nella funzione *loop()*, Arduino rimarrà in attesa di ricevere tutti i dati seriali ed isolerà le stringhe che terminano col carattere '\n' tramite la nostra funzione *getSerialLine()*. A questo punto la singola stringa di messaggio verrà parserizzata dalle funzioni *getColorFromString()* e *getComponentFromString()*, create per ricevere le informazioni su quale componente di colore settare e a quale determinato valore. Ora possiamo pilotare i piedini di uscita con la funzione di sistema *analogWrite()* che imposterà un segnale PWM sul piedino corrispondente. Tutto questo è mostrato nel **Listato 6**.

CONCLUSIONI

In queste pagine abbiamo descritto come realizzare un'applicazione Android minimale che possiamo già usare; tuttavia per dotarla di funzionalità aggiuntive che la rendano più utilizzabile e soprattutto che ci permettano di ampliare le nostre conoscenze sullo sviluppo Android, mancano ancora alcune cose. Nella prossima puntata vedremo come inserire un menu, in che modo creare più di una activity e come utilizzare l'oggetto ListView; in questo modo doteremo la nostra applicazione di un datalogger Bluetooth con ricerca automatica dei dispositivi remoti attivi e la possibilità di impostare le variazioni della tonalità della barra a LED tramite un temporizzatore.

Listato 6

```
void loop()
{
   String strRec = getSerialLine();

   digitalWrite(BOARD_LED, HIGH);

   Serial.println("Rec:" + strRec + "\n");

   char colComponent = getComponentFromString(strRec);
   int colValue = getColorFromString(strRec);

   if(colComponent == 'R')
   {
     analogWrite(RED_LED, colValue);
   }
   else if(colComponent == 'G')
   {
     analogWrite(GREEN_LED, colValue);
   }
   else if(colComponent == 'B')
   {
     analogWrite(BLUE_LED, colValue);
   }
}
```

per il MATERIALE

Lo shield Bluetooth descritto in questo progetto (cod. FT1032M) costa 34,00 Euro; la scatola di montaggio comprende tutti i componenti, il modulo radio Roving Networks RN-42, la basetta forata e serigrafata nonché tutte le minuterie. Nello stesso progetto viene utilizzato un shield RGB (cod. RGB_SHIELD) che costa 12,00 Euro nonché una board Arduino UNO, Euro 24,50. Il modulo Roving Networks RN-42 è anche disponibile separatamente al prezzo di 21,00 Euro. Tutti i prezzi si intendono IVA compresa.

Il materiale va richiesto a:
Futura Elettronica, Via Adige 11, 21013 Gallarate (VA)
Tel: 0331-799775 • Fax: 0331-792287 - www.futurashop.it

Riprendiamo lo shield Bluetooth per completare la nostra applicazione dotandola di ulteriori caratteristiche e funzionalità e approfittando per introdurre nuovi concetti Android. Quinta puntata.

di Andrea Chiappori

5

PROGRAMMIAMO CON ANDROID

Nella puntata precedente abbiamo analizzato il firmware per Android e per Arduino che permette di variare la luminosità ed il colore di una striscia a LED RGB ad interfaccia I²C-Bus. Immaginiamo ora di volere che la nostra striscia (o anche un singolo LED RGB) possa passare da un colore all'altro ad intervalli prestabiliti ed in tempi più o meno rapidi; questo potrebbe essere utile, ad esempio, per creare effetti alba-tramonto in presepi o modellini, senza dover ricorrere a più costose centraline dedicate. Dovremo perciò aggiungere alla nostra applicazione una modalità di programmazione tramite cui scegliere la sequenza di colori che invieremo ad Arduino tramite Bluetooth. Quest'ultimo memorizzerà i dati nella propria EEPROM, così sarà in grado di riprodurre la sequenza anche una volta che la connessione Bluetooth sarà terminata.

CREARE IL MENU

Per non appesantire ulteriormente il layout grafico della schermata principale, gestiremo la possibilità di programmare questa sequenza in una *View* dedicata, che sarà pertanto richiamabile tramite una voce di menu. Come saprete, uno dei tasti "fisici" in uno smartphone Android è quello che permette di mostrare nella parte bassa dello schermo un menu contestuale con più opzioni, in modo da arricchire l'applicazione con maggiori informazioni o configurazioni. Esistono due modi per creare i menu: uno "statico", che consiste nel ricorrere a un file xml, e l'altro dinamico, in cui si possono aggiungere o togliere voci di menu direttamente dal codice. Sono modi equivalenti nel risultato, ma quello dinamico ci permette di modificare le opzioni che compariranno nel menu anche a run-time, a seconda di determinate azioni compiute dall'utente. Vedremo comunque entrambe le soluzioni.

BtclientActivity.java *menuseq.xml

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

    <item android:id="@+id/scan"
          android:title="Scan Device"
          android:icon="@drawable/icon_scan" />

    <item android:id="@+id/seq"
          android:title="New Sequence"
          android:icon="@drawable/icon_seq" />

    <item android:id="@+id/abt"
          android:title="About"
          android:icon="@drawable/icon_information" />
</menu>
```

Fig. 1

Innanzi tutto dovremo aggiungere i seguenti *import* nel nostro codice java, per poter utilizzare le librerie interne adibite alla gestione del menu.

```
import android.view.Menu;
import android.view.MenuItem;
import android.view.MenuInflater;
```

A questo punto possiamo riprendere il codice della puntata precedente ed all'interno dell'Activity principale (class *BtClientActivity*) ridefinire la classe *onCreateOptionMenu()* in modo da poter gestire un menu. Ogni volta che l'applicazione ridisegnerà la *View* principale verrà chiamata questa funzione; sarà proprio in questo punto che andremo a scrivere il nostro codice.
Nel caso come quello in oggetto, in cui sappiamo che le voci del menu saranno sempre le stesse, possiamo scrivere un file xml come quello visibile in **Fig. 1**, che salveremo nella cartella layout. Come si può notare, ogni tag *<item>* rappresenta una voce di menu e ad ogni *item* possiamo associare un *Id* univoco, una stringa di testo (nel campo *android:title*) ed il percorso della relativa immagine (nel campo *android:icon*) che deve essere presente nel progetto, normalmente all'interno della cartella *drawable*.
Ricordiamo inoltre, come già visto in precedenza nel caso di risorse grafiche, di non specificare l'estensione dell'icona (che nel nostro caso è **.png**) nel nome del percorso e di fare uso solamente di lettere minuscole.
A questo punto abbiamo il nostro menu, ma è ancora "scollegato" e non visibile nella nostra applicazione. Per farlo comparire sullo schermo ogni qualvolta venga premuto il tasto "fisico" di menu, occorre scrivere ancora le poche righe di codice mostrate nel **Listato 1**, attraverso le quali istruiremo Android affinché possa utilizzare il file xml precedentemente creato, specificando il percorso della risorsa che sarà, appunto, *R.layout.menuseq*. Anche in questo caso *menuseq* è il nome del file xml privo della sua estensione e presente all'interno della cartella *layout*.
Il risultato che otterremo è visibile in **Fig. 2**.
Possiamo anche ottenere lo stesso risultato senza scrivere alcun file xml, ma modificando la funzione *onCreateOptionsMenu()* vista prima, come mostrato nel **Listato 2**.

Listato 1

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

Fig. 2

Sebbene i due metodi siano equivalenti, se utilizzate lo stesso menu anche in altre eventuali *View* vi consigliamo di usare il primo, perché vi risparmia di ripetere lo stesso codice più volte.
Inoltre, tramite il file xml è possibile aggiungere con semplicità maggiori caratteristiche, sfruttando il *code-assist* di Eclipse.
Qualunque metodo sia stato seguito, dobbiamo ora associare la pressione della voce di menu scelta (detta anche *MenuItem*) con la corrispondente azione da eseguire: questo viene realizzato ridefinendo la funzione *onOptionsItemSelected()* nella quale gestiremo appunto il parametro item.
Tale funzione, esplicitata nel **Listato 3**, effet-

Listato 2

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
        menu.add(Menu.NONE, 0, 0, "Scan").setIcon(getResources().getDrawable(R.drawable.icon_scan));
        menu.add(Menu.NONE, 1, 1, "New Seq").setIcon(getResources().getDrawable(R.drawable.icon_seq));
        menu.add(Menu.NONE, 2, 2, "About").setIcon(getResources().getDrawable(R.drawable.icon_
information));
        return super.onCreateOptionsMenu(menu);
}
```

tua un controllo sull'*Id* della voce di menu selezionata (che viene restituito dall' API di sistema *getItemId()*); in base al suo valore effettueremo le nostre azioni, chiamando le rispettive funzioni.
Adesso bisogna fare una piccola precisazione a proposito della funzione *getItemId()*, la quale ritorna un valore intero: nel caso avessimo utilizzato il metodo dinamico per creare il nostro menu, tale valore corrisponderà a quello specificato nel metodo Add (**Listato 2**) e quindi, nel nostro caso, il valore 0 corrisponderà alla prima voce di menu ("Scan"), il valore 1 alla seconda e così via. Se invece avessimo creato il menu utilizzando il file xml *menuseq.xml*, allora la lettura del codice risulterebbe semplificata per via della stringa inserita nel campo *android:id* del relativo item (**Fig. 1**) che ci permette di riscrivere il codice del **Listato 3**, utilizzando il prefisso **R** seguito dal nome dell'*Id*.
Il tutto sarà più chiaro leggendo il codice così trasformato, nel **Listato 4**. In particolare, abbiamo utilizzato tre voci di menu: la prima per effettuare la scansione dei dispositivi Bluetooth circostanti, la seconda per creare le nostre sequenze di luce ed infine la terza per aggiungere ulteriori informazioni. È sempre buona norma, infatti, dotare le nostre applicazioni di un menu, sia per non appesantire la *View* principale con troppi pulsanti, sia per poter aggiungere, oltre alle eventuali finestre di opzione, anche una finestra di dialogo con le informazioni sulle nostre generalità, il nostro logo o semplicemente la versione dell'applicazione.
A riguardo rimandiamo al riquadro "Finestre di dialogo Android" che trovate nella pagina seguente, dove si utilizza la classe Android *AlertDialog* per visualizzare le informazioni aggiuntive accessibili con la terza voce del menu.
Andiamo ora ad implementare le altre due voci di menu che ci permetteranno di acquisire nuovi concetti riguardanti il mondo Android.

CREARE NUOVE ACTIVITY

Finora in questa applicazione abbiamo utilizzato una singola Activity; in prima istanza, possiamo considerare un'Activity come la finestra grafica che permette all'utente di interagire con l'applicazione. Fisicamente un'Activity è una classe contenuta in un file sorgente con estensione **.java** (nel nostro

Listato 3

```
@Override
public boolean onOptionsItemSelected(MenuItem item)
{
        super.onOptionsItemSelected(item);
        switch(item.getItemId())
        {
                case 0:
                startScannerActivity();
                break;
                case 1:
                openSequencerView();
                break;
                case 2:
                openAboutDialog();
                break;
        }
        return true;
}
```

Listato 4

```
@Override
public boolean onOptionsItemSelected(MenuItem item)
{
        super.onOptionsItemSelected(item);
        switch(item.getItemId())
        {
                case R.id.scan:
                startScannerActivity();
                break;
                case R.id.seq:
                openSequencerView();
                break;
                case R.id.abt:
                openAboutDialog();
                break;
        }
        return true;
}
```

FINESTRE DI DIALOGO ANDROID

Abbiamo già visto in precedenza come realizzare un messaggio a comparsa tramite la classe *MessageToast*, ma Android mette a disposizione anche finestre di dialogo più complesse e facilmente gestibili, che possiamo utilizzare per visualizzare informazioni veloci come quelle in una classica finestra di "About". Si tratta della classe *AlertDialog*, la quale con pochi metodi, come quelli elencati qui di seguito, permette di ottenere il risultato di **Fig. 3** senza costringerci ad utilizzare un'altra Activity.

```
AlertDialog dlg = new AlertDialog.Builder(this).create();
dlg.setTitle("RGB Controller");
dlg.setMessage("Elettronica In\n Copyright ® 2012");
dlg.setIcon(R.drawable.icon_main);
dlg.show();
```

Fig. 3

Inoltre questa finestra verrà correttamente "distrutta" semplicemente premendo il tasto fisico "Back" sul dispositivo Android, senza doverci preoccupare di dotarla di un pulsante di uscita.

caso, *BtSequencerActivity.java*) che andremo a creare all'interno della cartella *src*, al pari degli altri file sorgente. Quando l'applicazione richiede particolari configurazioni o funzionalità aggiuntive, è meglio servirsi di nuove Activity per alleggerire la complessità, distribuendola su più finestre.
La principale differenza che troviamo rispetto al mondo Windows o Linux, al quale siamo più abituati, è che può essere attiva una sola Activity alla volta; nel caso di due o più Activity presenti nella nostra applicazione, se una di esse è in primo piano, le altre, oltre a non essere visibili, vengono configurate da Android in uno stato di secondo piano (o *background*) e, continuando a mantenere il loro ciclo di vita, potranno riprendere il loro stato di attività in futuro.
Senza scendere troppo nei dettagli, vediamo come poter lanciare una seconda Activity da quella principale. Occorre a questo punto introdurre il concetto di **Intent,** che può essere

Listato 5

```
private void openSequencerView()
{
Intent  sequencerIntent = new Intent(this, BtSequencerActivity.class);
          startActivity(sequencerIntent);
}
```

visto come un messaggio col quale riusciamo ad ordinare ad Android l'esecuzione di una generica azione di sistema.
In questo caso dovremo creare un nuovo Intent passandogli l'informazione sulla nuova classe Activity ed utilizzare il metodo *startActivity()* al quale passeremo come argomento proprio l'Intent. Tutto questo è visibile nel **Listato 5**, in cui viene implementata la funzione chiamata dopo aver premuto la seconda voce del menu (*New Seq*).
L'Activity che andremo a creare sarà una classe estesa della super classe Activity e pertanto dovremo obbligatoriamente implementare i metodi ereditati da essa. In **Fig. 4** è possibile vedere un template minimale della nostra Activity in cui i metodi, che rappresentano il passaggio tra i suoi vari stati e quindi il suo ciclo di vita, non fanno altro (almeno per ora) che richiamare le funzionalità della sua super classe.

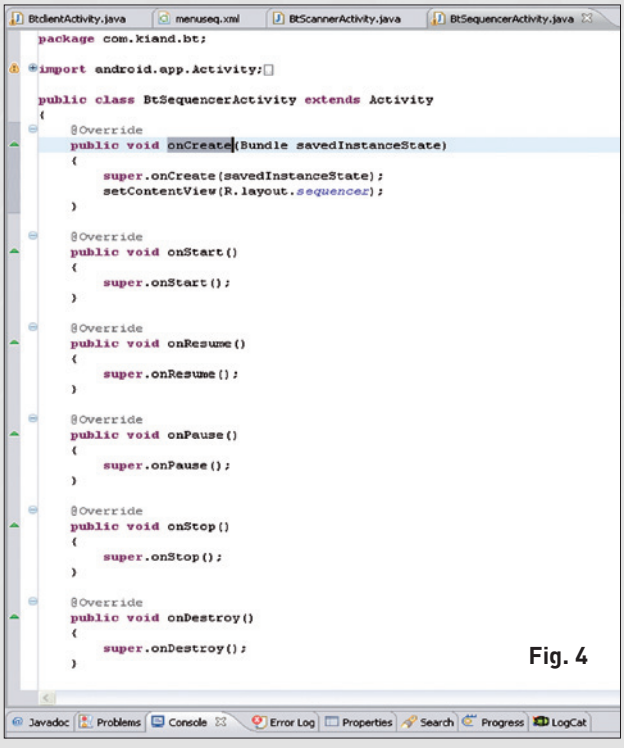


Fig. 4

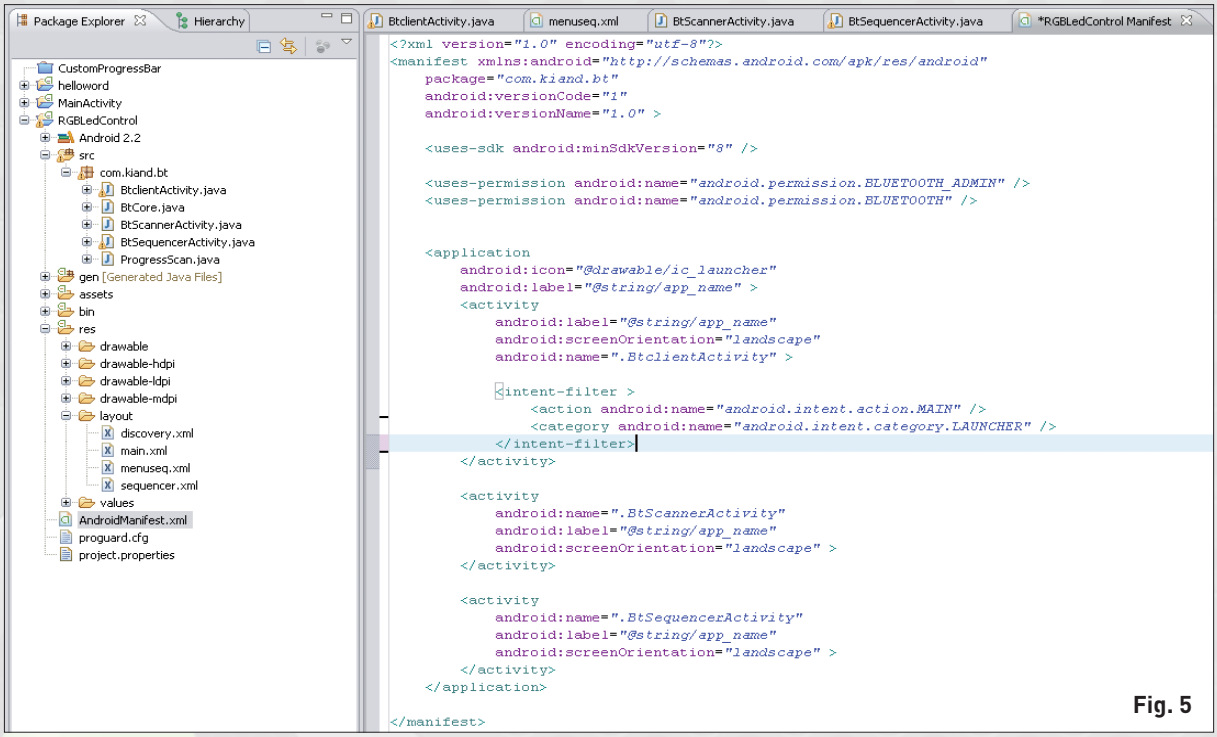


Fig. 5

Tutti questi metodi (*onCreate()*, *onStart()*, *onResume()*, *onPause()*, *onStop()*, *onDestroy()*) vengono chiamati direttamente dal sistema operativo e noi possiamo gestirli in modo attivo inserendo codice personalizzato a seconda delle nostre esigenze, oppure demandare tutto alla super classe tramite, ad esempio, la riga di codice *super.onStop()* valida nel caso del metodo *onStop()*.
Possiamo spiegare brevemente questi metodi dicendo che la *onCreate()* viene chiamata una sola volta, non appena viene eseguita la *startActivity()* vista precedentemente. È proprio qui che andremo ad impostare il layout grafico scelto per questa Activity (tramite la riga *setContentView(R.layout.sequencer.xml)* e ad aggiungere il nostro codice per associare le azioni agli eventi generati dalla pressione di tasti e controlli generici.
Il metodo *onStart()* viene chiamato anch'esso una sola volta e poco prima che l'Activity venga fisicamente disegnata. Il metodo *onResume()* viene chiamato prima che l'Activity passi in stato *running*, quindi verrà chiamato la prima volta, ma anche ogni volta che l'Activity esca dallo stato *stopped* e torni attiva.
Il metodo *onPause()* viene chiamato quando l'Activity è ancora visibile, ma un'altra Activity sta prendendo il suo posto per andare in *foreground*, ed il metodo *onStop()* quando proprio l'Activity non è più visibile. Infine, il metodo *onRestart()* è chiamato ogni qualvolta l'Activity venga ridisegnata ed il metodo *onDestroy()* appena prima che l'Activity venga distrutta.
Per poter utilizzare l'Activity appena creata, non dobbiamo dimenticarci di informare Android della sua esistenza; allo scopo la aggiungiamo nel file *AndroidManifest.xml*, già visto ed utilizzato nel corso delle puntate precedenti, che si presenterà come in **Fig. 5**. Da notare il tag <*intent-filter*> che deve essere presente solo nell' Activity principale, con la quale parte l'applicazione e dalla quale verranno lanciate le altre Activity.
Spiegate le generalità, vediamo ora come useremo le Activity in questa puntata.

LIGHT SEQUENCER

Premendo sulla voce di menu New Seq precedentemente creata, apparirà ora una nuova finestra con i controlli che abbiamo aggiunto nel nuovo file di layout *sequencer.xml*. e visibile in **Fig. 6**. Come noterete, sono gli stessi controlli usati nell'Activity principale, ma oltre a questi ci sono anche altri pulsanti

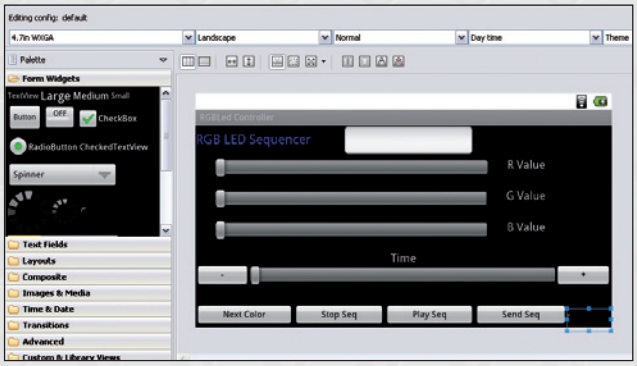


Fig. 6

per consentire maggiori funzionalità. L'idea è quella di creare una sequenza di colori direttamente sul nostro smartphone, quindi inviarla ad Arduino sotto forma di stringa Bluetooth. Abbiamo inserito anche una ulteriore *SeekBar* attraverso la quale settare il tempo (in secondi) in cui resterà attivo il colore scelto. Cliccando poi sul pulsante "Next Color" il colore attivo ed il relativo intervallo verranno memorizzati in una lista di stringhe e potremo scegliere un altro colore col suo relativo intervallo. Una volta terminata la nostra sequenza possiamo premere il pulsante "Stop Seq" e successivamente il pulsante "Play Seq" per riprodurla mediante l'RGB shield, oppure "Send Seq" per inviarla ad Arduino, che la memorizzerà sulla propria EEPROM. A questo punto, premendo "Play Seq" nella finestra principale la sequenza verrà rieseguita con un loop infinito anche dopo aver terminato l'applicazione sullo smartphone. Per l'invio della sequenza possiamo inventarci un protocollo di messaggi come preferiamo; sapendo che non abbiamo vincoli di tempo stringenti utilizzeremo semplicemente l'invio di stringhe.
Sulla pressione del pulsante "Send Seq" abbiamo, così,scelto di inviare il carattere 'S' seguito dal numero totale di colori e successivamente le informazioni sulle componenti di colore e l'intervallo di tempo tramite i caratteri R, G,B e T.
Se, ad esempio, volessimo una sequenza che inizi con un colore rosso per poi passare dopo 30 secondi ad un verde e dopo 40 al blu, e desiderassimo rieseguire il loop dopo 50 secondi, invieremo un messaggio di questo tipo:

S3
R255 G000 B000 T30
R000 G255 B000 T40
R000 G000 B255 T50

Dall'altra parte Arduino, dovrà essere in grado di parserizzare le stringhe correttamente, salvare i valori in un array, memorizzarli internamente e utilizzarli quando, dalla nostra applicazione Android, riceverà la stringa "P0".

Listato 6

```
<SeekBar
        android:id="@+id/seekbarB"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:max="255"
        android:progress="0"
        android:progressDrawable="@drawable/custom_seek_bar"
        android:thumb="@drawable/custom_seek_bar_thumb"
        android:thumbOffset="0dip"
        android:maxHeight="10dip"
        android:layout_weight="80"
/
```

CONTROLLI PERSONALIZZATI

Non ci addentreremo nel codice dell'Activity *BtSequencerActivity* perché molto simile a quello dell'Activity principale già vista, ma possiamo approfittare dell'occasione per darvi alcuni suggerimenti su come personalizzare le vostre applicazioni creando controlli custom che presentino un' interfaccia grafica a vostro piacimento, un po' diversa da quelle classiche. Prendiamo, ad esempio, le quattro *SeekBar* inserite nel layout in **Fig. 6** ed immaginiamo di volerne cambiare alcuni parametri estetici come la forma del cursore o il colore di riempimento della barra. Questa, per impostazione predefinita è arancione, ma nel nostro caso sarebbe interessante fosse rossa per la *SeekBar* relativa alla componente rossa, verde per la componente verde e blu per la componente blu. Android ci offre questa

Listato 7

```
<selector xmlns:android="http://schemas.android.com/
apk/res/android">

    <item android:state_pressed="true"
      android:drawable="@drawable/custom_thumb_state_pressed" />

    <item android:state_focused="true"
      android:drawable="@drawable/custom_thumb_state_selected" />
    <item android:state_selected="true"
      android:drawable="@drawable/custom_thumb_state_selected" />
    <item android:drawable="@drawable/custom_thumb_state_default" />
</selector>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<layer-list
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@android:id/background">
        <shape>
            <corners
                android:radius="10dip" />

            <gradient
                android:startColor="#ff00ff00"
                android:centerColor="#0000ff00"
                android:endColor="#ff000000"
                android:centerY="0.5"
                android:angle="270" />
        </shape>
    </item>

    <item
        android:id="@android:id/progress">
        <clip>
            <shape>
                <corners
                    android:radius="10dip" />
                <gradient
                    android:startColor="#ffbf0dff"
                    android:endColor="#fffef5555"
                    android:angle="90" />
            </shape>
        </clip>
    </item>
</layer-list>
```

Fig. 7

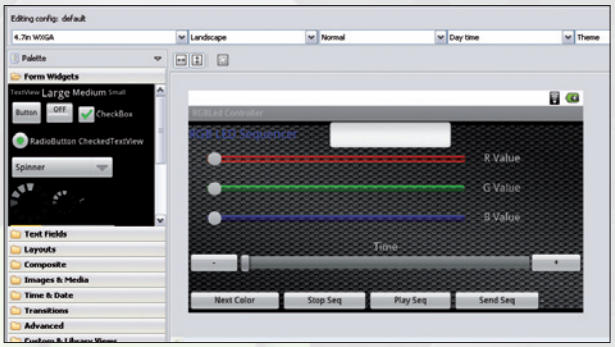


Fig. 8

barra a sinistra del cursore (*progress*). Ognuno di questi item è personalizzabile tramite una serie di tag, come *corners* per definire gli angoli della barra, *solid* per definire un colore di sfondo pieno, piuttosto che *gradient* per definire l'effetto di sfumatura desiderato.
Allo stesso modo possiamo modificare la forma e il colore del cursore, aggiungendo il campo *android:thumb* ed impostandolo un file di risorsa come quello nel **Listato 7**, che a sua volta presenta collegamenti ad altri file di risorsa in base ai differenti stati del cursore (*pressed*, *focused*, *selected*).
Possiamo allora sbizzarrirci su forma, colore ed altri effetti, creando tanti file xml o immagini per costruire tre *SeekBar* di colori diversi e con caratteristiche estetiche differenti, come si può vedere in **Fig. 8**.
A questo punto, sulla base delle informazioni viste prima, potremmo anche creare un nostro controllo e dargli un nome a nostra scelta, semplicemente creando un nuovo file xml (non importa il nome del file, basta che sia presente il tag <*resources*>) e che andremo a mettere all'interno della cartella *values*. Un file di questo tipo è visibile nel **Listato 8**.
Ora nel nostro file di layout potremmo usare direttamente questo controllo usando il tag con lo stesso nome che gli abbiamo dato in

Listato 8

```
<?xml version="1.0" encoding="UTF-8"?>
<resources>
        <style name="CustomSeekBar" parent="android:Widget.SeekBar">
                <item name="android:progressDrawable">@drawable/custom_seek_bar_r</item>
                <item name="android:thumb">@drawable/custom_seek_bar_thumb</item>
                <item name="android:thumbOffset">0dip</item>
        </style>
</resources>
```

possibilità in modo tutto sommato semplice: riprendiamo il file di layout *sequencer.xml* ed aggiungiamo all'interno del tag <*SeekBar*> che ci interessa modificare, il campo *android:progressDrawable*, che imposteremo con un file di risorsa personalizzato (vedere il **Listato 6**).
Questo file di risorsa (*@drawable/custom_seek_bar_R*) è un file xml (nulla, però, ci vieta di usare un'immagine) come quello visibile in **Fig. 7** in cui sono presenti due item: uno per ridisegnare lo sfondo della barra di scorrimento (*background*) e l'altro per ridisegnare la precedenza (nel nostro caso <*CustomSeekBar*>) preoccupandoci solo delle dimensioni e della sua posizione all'interno della *View*.

SCANSIONE BLUETOOTH

Negli esempi fatti fino ad ora la comunicazione Bluetooth tra Android ed Arduino è avvenuta cablando nel codice l'indirizzo MAC del modulo RN-42; ciò rende l'applicazione poco flessibile e nel caso in cui volessimo riutilizzare questo stesso codice per gestire un altro modulo Bluetooth ci troveremmo fortemente vincolati.

Ora che abbiamo introdotto il menu, possiamo lanciare un'Activity dedicata che gestisca la ricerca dei dispositivi remoti nelle vicinanze e restituisca il MAC Address del dispositivo selezionato. La prima voce di menu, "Scan Device" fa proprio questo. Essendo un'Activity da cui vogliamo ottenere un risultato che utilizzeremo nell'Activity principale, dovremmo lanciarla in un modo diverso dal precedente e quindi, anziché utilizzare il metodo *startActivity()*, useremo *startActivityForResult()* (**Listato 9**).
Sempre nell'Activity principale, dovremo gestire questo risultato quando il sistema operativo chiamerà il metodo *onActivityResult()* (vedere ancora il **Listato 9**) in cui ricaviamo il MAC Address dal parametro data.
Analizziamo ora la classe *BtScannerActivity* che ha poche righe di codice ma è molto particolare.
Nel metodo *onCreate()* inseriremo il solito file di layout *discovery.xml*, che questa volta avrà un controllo nuovo: la lista *ListView*, in cui verranno elencati i dispositivi rilevati. Sempre nella *onCreate()*, troviamo la creazione dinamica di un controllo *ProgressScan* (il circoletto rotante che in Android indica l'attesa dell'esecuzione di un'operazione) attraverso la quale viene lanciato un thread parallelo *(_discoveryWorker)* che effettua la scansione Bluetooth vera e propria.
Precisiamo che *_discoveryWorker* è un oggetto della classe base *Runnable* creato nello stesso file e che in Android viene utilizzato per gestire appunto i thread, implementando il metodo *run()* di questa classe.
Infine troviamo i due **IntentFilter**, discoveryFilter e foundFilter (visibili nel **Listato 10**) che sono i responsabili della rilevazione dei dispositivi Bluetooth. Un IntentFilter può essere visto come una sorta di direttiva con cui informiamo il sistema operativo su quali Intent (messaggi provenienti dal sistema operativo) siamo in grado di gestire e con quale oggetto di classe.
Nel nostro caso l'Intent *foundFilter* sarà attivo sulla ricezione del messaggio Bluetooth *ACTION_FOUND*, e quindi ogni qual volta venga rilevato un nuovo dispositivo questo sarà gestito nel metodo *onReceive()* dell' oggetto *_foundReceiver*, dal momento che, tramite il metodo *registerReceiver()*, abbiamo associato l'intent a questo oggetto. Nel metodo *onReceive()* inseriremo il nuovo dispositivo nella lista grafica, che così verrà continuamente aggiornata (*ShowDevices()*).
Per far terminare correttamente il thread e quindi la scansione, useremo proprio il secondo IntentFilter (*discoveryFilt er*) che, invece, gestirà il messaggio Bluetooth di fine scansione (*ACTION_DISCOVERY_FINISHED*) facendo terminare il thread *_discoveryWorker* e lasciando all'utente la scelta del dispositivo.
A questo punto il controllo passerà alla *onListItemClick()* (sempre nel **Listato 10**) in cui saremo noi a creare un messaggio (Intent) con il MAC Address del dispositivo scelto, che restituiremo all'Activity principale.

Listato 9

```
public void startScannerActivity()
{
        Log.v(TAG, "Start Scan");
        Intent  btScanIntent = new Intent(this, BtScannerActivity.class);
        startActivityForResult(btScanIntent, BtCore.REQUEST_DISCOVERY);
}
// after select, connect to device
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
        if (requestCode != BtCore.REQUEST_DISCOVERY)
        {
                return;
        }
        if (resultCode != RESULT_OK)
        {
                return;
        }
        BluetoothDevice device = data.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
        BtCore.serverAddressAfterScan =  device.getAddress();
}
```

Listato 10

```
protected void onCreate(Bundle savedInstanceState)
{
        super.onCreate(savedInstanceState);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_BLUR_BEHIND,WindowManager.LayoutParams.FLAG_BLUR_BEHIND);
        setContentView(R.layout.discovery);
        Log.w("RGB-CONTR-SCAN", "onCreate");

        // BT isEnable
        if (!_bluetooth.isEnabled())
        {
                Log.w("RGB-CONTR-SCAN", "Disable!");
                finish();
                return;
        }
        // Register Receiver
        IntentFilter discoveryFilter = new IntentFilter(BluetoothAdapter.ACTION_DISCOVERY_FINISHED);
        registerReceiver(_discoveryReceiver, discoveryFilter);
        IntentFilter foundFilter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
        registerReceiver(_foundReceiver, foundFilter);
        ProgressScan.indeterminateInternal(BtScannerActivity.this, _handler, "Scanning...", _discoveryWorker,
                                new OnDismissListener()
                                                {
                                                        public void onDismiss(DialogInterface dialog)
                                                        {
                                                                for (; _bluetooth.isDiscovering();)
                                                                {
                                                                        _bluetooth.cancelDiscovery();
                                                                }
                                                                _discoveryFinished = true;
                                                        }
                                                }, true);
}
private Runnable _discoveryWorker = new Runnable()
{
        public void run()
        {
                // Start search device
                _bluetooth.startDiscovery();
                Log.d("RGB-CONTR-SCAN", "Starting Discovery");
                for (;;)
                {
                        if (_discoveryFinished)
                        {
                                Log.d("RGB-CONTR-SCAN", "Finished");
                                break;
                        }
                        try
                        {
                                Thread.sleep(100);
                        }
                        catch (InterruptedException e)
                        {
                        }
                }
        }
};
protected void onListItemClick(ListView l, View v, int position, long id)
{
        Log.d("RGB-CONTR-SCAN", "Click device");
        Intent result = new Intent();
        result.putExtra(BluetoothDevice.EXTRA_DEVICE, _devices.get(position));
        setResult(RESULT_OK, result);
        finish();
}
```

CONCLUSIONI

Abbiamo completato la nostra applicazione Bluetooth rendendola più completa e funzionale ed acquisendo alcuni concetti Android (*Intent, FilterIntent, Runnable*) che ci troveremo ad utilizzare anche nei nostri prossimi progetti. Durante le puntate seguenti analizzeremo come gestire da un'applicazione Android, la ricezione e l'invio di messaggi e chiamate telefoniche verso dispositivi remoti dotati di modulo cellulare.

Presentiamo una nuova applicazione in grado di gestire chiamate GSm ed SMS, affrontiamo la creazione di un servizio Android ed approfondiamo l'utilizzo di alcuni sensori del nostro smartphone, come accelerometro e localizzatore GPS. Sesta puntata.

di Andrea Chiappori

6

PROGRAMMIAMO CON ANDROID

Dopo aver spiegato l'interazione tra il modulo Bluetooth RN-42 e Android, ora vedremo come gestire direttamente chiamate e messaggi GSM e in che modo interagire con i sensori di movimento e localizzazione dello smartphone allo scopo di pilotare lo shield Arduino GSM presentato nelle puntate precedenti. In particolare, dopo questa lezione saremo in grado di attivare dispositivi esterni collegati allo shield, tramite una chiamata gestita internamente dalla nostra applicazione, ed impartire diversi comandi grazie ad SMS personalizzati.
Potremo dotare la nostra applicazione di maggiori funzionalità: ad esempio far partire una chiamata verso un numero telefonico prestabilito in seguito ad uno "scuotimento" ripetuto dello smartphone, oppure appena questo si troverà in prossimità di una certa zona geografica, anch'essa impostabile (**Fig. 1**). Questo ci dà l'occasione di fare una panoramica sui sensori di movimento e sul localizzatore GPS di Android, ma anche di introdurre un nuovo concetto Android che è il "servizio", il quale ci risparmia di dover tenere sempre attiva e a pieno schermo la nostra applicazione durante il suo ciclo di vita. Anche se questa applicazione nasce per scopo didattico, potrà comunque tornarci utile per realizzare automatismi comandati a distanza, come ad esempio un apricancello o un avviso di presenza.

SERVIZIO ANDROID

Mentre nell'applicazione descritta nel fascicolo precedente (*RGBLedController*) abbiamo sempre avuto a che fare con le Activity, che sono classi dotate di un'interfaccia grafica verso l'utente e con le quali è possibile interagire attivamente, questa volta occorre fare in modo che il nostro codice funzioni in background,

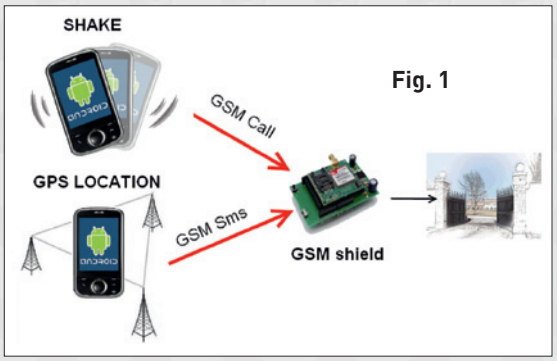
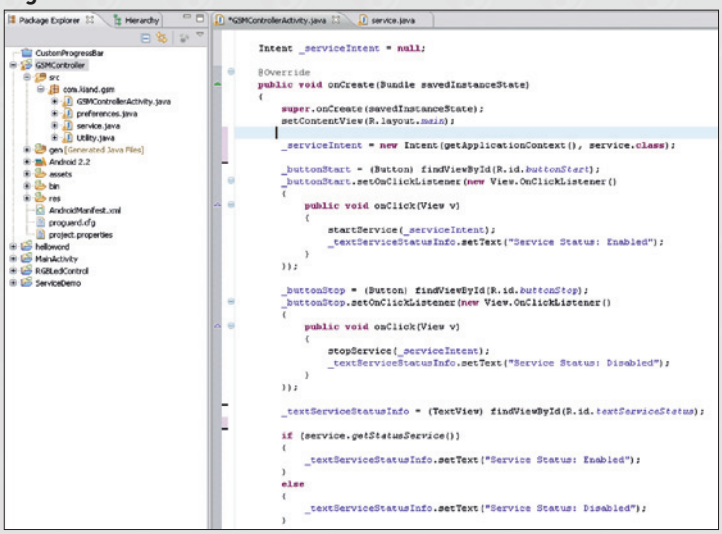


Fig. 1

permettendoci di utilizzare il nostro dispositivo normalmente, ma rimanendo comunque in attesa di alcuni eventi esterni (come segnali da sensori di movimento, luminosità, GPS ed altri) ai quali esso reagirà di conseguenza.
Per fare ciò useremo appunto un servizio, che in Android altro non è che una classe (e quindi un file) estesa della classe base *Service*.
Prima di addentrarci nella creazione del nostro primo servizio, creiamo un nuovo progetto seguendo gli stessi passi delle puntate precedenti, anche perché la prima classe con cui viene lanciata l'applicazione sarà sempre una Activity (quindi dotata di interfaccia grafica) attraverso la quale potremo avviare il servizio vero e proprio, arrestarlo e soprattutto configurarlo a nostro piacimento, ad esempio per cambiare i numeri di telefono di destinazione, la sensibilità dei sensori ed altre opzioni che vedremo più avanti.
Quindi, inizialmente aggiungeremo due semplici pulsanti Start e Stop con cui avvieremo e arresteremo il servizio, ed una *TextView* che ci indicherà lo stato del servizio (**Fig.1**).
Senza scendere nei dettagli, assoceremo alla pressione di due pulsanti le funzioni di sistema con cui potremo lanciare e arrestare il servizio dall'Activity.
Come visibile in **Fig. 2**, per avviare il servizio occorre creare, sempre nell'Activity principale, un Intent al quale passeremo il nome della nostra classe *GSMService* (che poi è lo stesso nome del file *GSMService.java*) e che aggiungeremo al progetto nella sottocartella *src*.
Successivamente chiameremo il metodo *startService()* passandogli come parametro l'Intent appena creato. Analogamente, per fermare il servizio useremo il metodo *stopService()* usando sempre lo stesso Intent.
Soffermiamoci ora sulla scrittura del nostro servizio, ovvero del file *GSMService.java*.
Per facilitare la comprensione abbiamo aggiunto i messaggi Toast all'ingresso delle funzioni che gestiscono il servizio e che ci segnalano il ciclo di vita del servizio.
In **Fig. 3** è mostrato lo scheletro di un servizio che ora andremo a completare secondo le nostre esigenze. In seguito al lancio del servizio tramite *startService()*, questo viene prima creato e subito dopo avviato. In queste fasi (rispettivamente metodo *onCreate()* e *onStart()*) inseriremo il codice per permettere lo svolgimento delle azioni che vogliamo siano eseguite in background.
Solitamente andremo a registrare alcune particolari classi dette "Listener" che si metteranno in ascolto di particolari eventi del sistema operativo e che vedremo in seguito.
Il metodo *onDestroy()* viene chiamato quando il servizio termina in seguito alla chiamata del metodo *stopService()* dell'Activity, oppure a quella

Fig. 2

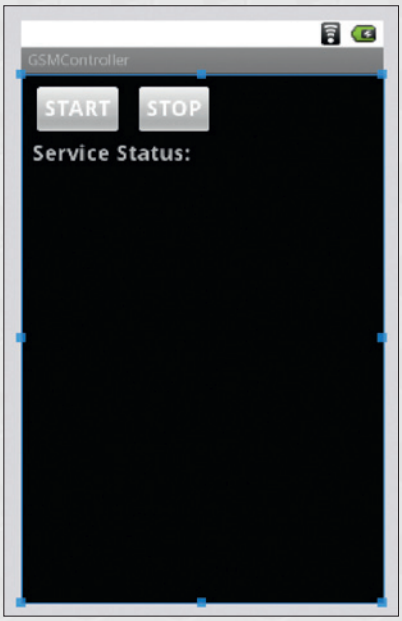


```
Intent _serviceIntent = null;

@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    _serviceIntent = new Intent(getApplicationContext(), service.class);

    _buttonStart = (Button) findViewById(R.id.buttonStart);
    _buttonStart.setOnClickListener(new View.OnClickListener()
    {
        public void onClick(View v)
        {
            startService(_serviceIntent);
            _textServiceStatusInfo.setText("Service Status: Enabled");
        }
    });

    _buttonStop = (Button) findViewById(R.id.buttonStop);
    _buttonStop.setOnClickListener(new View.OnClickListener()
    {
        public void onClick(View v)
        {
            stopService(_serviceIntent);
            _textServiceStatusInfo.setText("Service Status: Disabled");
        }
    });

    _textServiceStatusInfo = (TextView) findViewById(R.id.textServiceStatus);

    if (service.getStatusService())
    {
        _textServiceStatusInfo.setText("Service Status: Enabled");
    }
    else
    {
        _textServiceStatusInfo.setText("Service Status: Disabled");
    }
```

Fig. 3

```
package com.kiand.gsm;

import android.app.Service;

public class GSMService extends Service
{
    @Override
    public IBinder onBind(Intent intent)
    {
        return null;
    }

    @Override
    public void onCreate()
    {
        Toast.makeText(this, "Service Created", Toast.LENGTH_LONG).show();
    }

    @Override
    public void onStart(Intent intent, int startid)
    {
        Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
    }

    @Override
    public void onDestroy()
    {
        Toast.makeText(this, "Service Stopped", Toast.LENGTH_LONG).show();
    }
}
```

Fig. 4



del metodo *stopSelf()* interno alla classe *Service*; in questa fase vengono anche deregistrate le classi Listener.
Infine il metodo *onBind()* nel nostro caso ritornerà sempre valore nullo, perché in questa applicazione non gestiremo servizi di tipo *Bounded* nei quali un'altra Activity può interfacciarsi al servizio.
Possiamo ora aggiungere alla nostra classe, anche un nostro metodo privato *getStatusService()* attraverso il quale controlleremo lo stato del servizio (attivo o arrestato) all'avvio dell'applicazione: si tratterà semplicemente di settare una nostra variabile interna a *true* nella fase di Start e a false nella fase di Destroy.
È bene, infine, ricordare che un servizio, in quanto tale, può essere attivo anche se l'Activity principale della nostra applicazione è stata chiusa; esso continuerà a funzionare e lo potremo arrestare riaprendo la nostra applicazione e premendo il pulsante Stop. Per verificare lo stato del servizio, useremo la casella di testo *textServiceStatusInfo*.
Un primo layout (*main.xml*) dell'applicazione è mostrato in **Fig. 4**.
Prima di continuare può essere utile una breve precisazione sulla gestione della RAM in un sistema Android: quando chiudiamo una qualsiasi applicazione Android, questa in realtà continuerà a rimanere in memoria fino a quando una successiva applicazione richiederà una quantità di memoria superiore

Listato 1

```
//  GSMService.java

@Override
public void onCreate()
{
        Toast.makeText(this, "Service Created", Toast.LENGTH_LONG).show();
        Log.d(TAG, "onCreate");

        // sensor
        _sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
        Sensor s = _sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

        _sensorManager.registerListener(_accelerometerSensorListener, s, SensorManager.SENSOR_DELAY_NORMAL);

}


@Override
public void onStart(Intent intent, int startid)
{
        Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
        Log.d(TAG, "onStart");

        _statusService = true;

}

@Override
public void onDestroy()
{
        Toast.makeText(this, "Service Stopped", Toast.LENGTH_LONG).show();
        Log.d(TAG, "onDestroy");

        _sensorManager.unregisterListener(_accelerometerSensorListener);
        _sensorManager = null;

        _statusService = false;


}
```

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.kiand.gsm"
    android:versionCode="1"
    android:versionName="1.0" >

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

    <uses-permission android:name="android.permission.CALL_PHONE" />
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.VIBRATE"/>
    <uses-permission android:name="android.permission.ACCESS_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_GPS"/>
    <uses-permission android:name="android.permission.ACCESS_MOCK_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:label="@string/app_name"
            android:name=".GSMControllerActivity"
            android:screenOrientation="portrait" >
            <intent-filter >
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service android:enabled="true" android:name=".GSMService" />

    </application>

</manifest>
```

Fig. 5

a quella rimasta libera, oppure finché non sarà trascorso un certo periodo di tempo dall'ultimo utilizzo.
Un servizio in background ha una priorità minore di un'Activity in primo piano, ma sempre maggiore di un'Activity che rimane in memoria dopo essere stata chiusa; in questo modo siamo sicuri che, salvo condizioni di sistema estremamente particolari, difficilmente Android interromperà il nostro servizio.
Così come fatto in precedenza, quando abbiamo aggiunto altre Activity al progetto, dobbiamo informare il sistema dell'utilizzo della classe *GSMService*, aggiungendo la seguente istruzione al file *AndroidManifest.xml*:
<service android:enabled="true" android name=". GSMService" />
Questa riga è fondamentale per la corretta esecuzione del servizio, altrimenti anche se la compilazione del progetto avrà ugualmente luogo, otterremo un errore a run-time.
Il file *AndroidManifest.xml* completo è visibile nella **Fig. 5**.
A questo punto, dopo aver lanciato il nostro servizio dall'Activity principale, possiamo anche chiudere l'applicazione (tramite il pulsante fisico *back*) ed usare il nostro cellulare normalmente, sapendo che potremo arrestarlo rilanciando l'applicazione e premendo il pulsante Stop.
Quando accederemo nuovamente all'applicazione, la casella di testo ci informerà sullo stato del servizio.
Adesso abbiamo un servizio, che tuttavia, per il momento, non fa ancora alcunché; per verificare la sua presenza nel sistema basta utilizzare una di quelle "*app manager*" scaricabili gratuitamente dal market Android (Google Play) o già presenti nel dispositivo, che sono in grado di monitorare tutti i servizi attivi sul tablet o smartphone.

GESTIONE DEI SENSORI ANDROID

Andiamo ora a "riempire" il servizio creato, facendo in modo che reagisca ad eventi di sistema come l'attivazione di sensori presenti nel nostro dispositivo. Per capire quanti e quali sensori sono presenti nel nostro smartphone, inseriamo le seguenti linee, per esempio, nel metodo *onCreate()* dell'Activity principale:

```
SensorManager manager = (SensorManager)
getSystemService(SENSOR_SERVICE);
List<Sensor> list = manager.getSensorList(Sensor.
TYPE_ALL);
```

La classe *SensorManager* rappresenta il gestore "globale" di tutti i sensori presenti. Dall'oggetto *manager* siamo in grado di ottenere un oggetto appartenente alla classe *Sensor* che rappresenta il nostro sensore vero e proprio. Il metodo *getSensorList()* dell'oggetto manager ci restituisce il sensore desiderato.
Avendo usato l'identificativo TYPE_ALL per indicare tutti i dispositivi, il *manager* ci restituirà la lista completa dei sensori presenti; se invece avessimo voluto cercare un solo sensore, avremmo potuto usare direttamente il tipo appropriato, come ad esempio TYPE_PRESSURE nel caso del sensore di pressione.
Nel nostro caso useremo il metodo *getDefaultSensor()* che restituisce direttamente il singolo oggetto *Sensor* corrispondente. Nel riquadro "Sensori Android" che trovate in queste pagine potete vedere con quali tipi poter accedere a tutti i sensori disponibili.
Di questi, nella nostra applicazione useremo l'accelerometro e quindi accederemo al nostro oggetto *Sensor* tramite il tipo TYPE_ACCELEROMETER. Dall'oggetto *Sensor* siamo anche in grado di ottenere alcune interessanti informazioni come la risoluzione, la potenza usata dal sensore, il raggio d'azione ed altre funzionalità.
Volendo ottenere tutti i dati dei sensori in modalità asincrona, senza perciò interrogarli a polling, occorrerà registrare un oggetto di classe Listener (*_accelerometerSensorListener*) attraverso il metodo *registerListener()* del nostro gestore *SensorManager*, come si vede nel **Listato 1**.
In questo modo stiamo indicando al sistema Android di intercettare tutti i dati disponibili del sensore e di notificarli alla nostra classe Listener

Listato 2

```
private  SensorEventListener _accelerometerSensorListener = new SensorEventListener()
{
          private int _count = 0;
          private long firstShake = 0;

          private int SHAKE_DURATION = 600;
          private int SHAKE_COUNT = 4;
          private int SHAKE_THRESHOLD = 2000;
          private int SHAKE_INTERVAL = 500;


          public void onAccuracyChanged(Sensor sensor, int accurancy)
          {
                              //...
          }
          public void onSensorChanged(SensorEvent event)
          {
                    float[] values = event.values;
                    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER)
                    {
                              long curTime = System.currentTimeMillis();

                              // update every 100ms.
                              if ((curTime - lastUpdate) > 100)
                              {
                                        long diffTime = (curTime - lastUpdate);
                                        lastUpdate = curTime;

                                        x = values[SensorManager.DATA_X];
                                        y = values[SensorManager.DATA_Y];
                                        z = values[SensorManager.DATA_Z];

                                        NumberFormat numberFormat = new DecimalFormat("0.00");
                                        String strX = numberFormat.format(x);
                                        String strY = numberFormat.format(y);
                                        String strZ = numberFormat.format(z);


                                        float speed = Math.abs(x+y+z - last_x - last_y - last_z) / diffTime
                                                                                  * 10000;

                                        if (speed > (SHAKE_THRESHOLD) )
                                        {

                                                  Log.d(TAG, "_count:" + _count );
                                                 if (_count == 0)
                                                 {
                                                           firstShake = curTime;
                                                 }
                                                 _count++;

                                                 long durationStep = (curTime - firstShake);
                                                 if(durationStep > SHAKE_INTERVAL)
                                                 {
                                                           _count = 0;
                                                           Log.d(TAG, "durationStep:" + durationStep);
                                                 }

                                                 if (_count >= SHAKE_COUNT)
                                                 {
                                                           _count = 0;
                                                           long duration = (curTime - firstShake);
                                                           if( duration <= SHAKE_DURATION )
                                                           {
                                                            Log.d(TAG, "SHAKE duration:" + duration + "count:"
                                                 + _count );

                                                            _vibrate.vibrate(500);
                                                            callNumber("3346710868");
                                                           }
                                                 }
                                        }
                                        last_x = x;
                                        last_y = y;
                                        last_z = z;
                              }
                    }

          }
};
```

SENSORI ANDROID

Android, attraverso la classe *SensorManager*, ci permette di accedere ai sensori presenti nel nostro smartphone o tablet. Non tutti i dispositivi Android hanno, però, stesse caratteristiche e i medesimi sensori, perciò può essere utile ottenere una lista dei sensori presenti tramite le seguenti righe di codice:

```
SensorManager manager = (SensorManager)
getSystemService(SENSOR_SERVICE);
List<Sensor> list = manager.getSensorList(Sensor.
TYPE_ALL);
```

Se invece sapessimo già con che tipo di sensore avremo a che fare, possiamo direttamente usare il metodo *getDefaultSensor()* passandogli il tipo corrispondente come nella seguente riga:

```
Sensor s = _sensorManager.getDefaultSensor(Sensor.
TYPE_ACCELEROMETER);
```

Tipo	Sensore
TYPE_GYROSCOPE	Giroscopio
TYPE_LIGHT	sensore di luce
TYPE_ACCELEROMETER	Accelerometro
TYPE_MAGNETIC_FIELD	sensore di campo magnetico
TYPE_PRESSURE	sensore di pressione
TYPE_ORIENTATION	sensore di orientazione
TYPE_PROXIMITY	sensore di prossimità
TYPE_TEMPERATURE	sensore di temperatura

Metodo sensore	Descrizione
getMaximumRange()	il raggio di azione del sensore
getname()	il nome del sensore
getPower()	la potenza consumata dal sensore in mA
getResolution()	la risoluzione del sensore
getVendor()	il produttore del modulo
getVersion()	la versione del modulo

Nelle tabelle in questo riquadro vedete la lista completa dei sensori ed alcuni metodi dell'oggetto *Sensor* tramite i quali siamo in grado di conoscere alcune interessanti informazioni.

che analizzeremo a breve. Da notare anche il terzo parametro passato al metodo *registerListener()* che rappresenta la frequenza di aggiornamento del sensore. Per la nostra applicazione abbiamo scelto un valore normale (*SENSOR_DELAY_NORMAL*), ma in alcuni casi potrebbe essere necessario un campionamento più alto (*SENSOR_DELAY_FAST*) o anche molto elevato (*SENSOR_DELAY_GAME*).
Come mostrato nel **Listato 1**, al momento della distruzione del servizio (metodo *onDestroy()*) occorre deregistrare la classe Listener, per evitare di lasciare la CPU in attività anche dopo la chiusura dell'applicazione.

Occupiamoci, adesso, della scrittura della classe *SensorEventListener* presente nel **Listato 2**, nella quale abbiamo accesso ai dati veri e proprio letti dal nostro sensore; in particolare, implementeremo il metodo *onAccuracyChanged()* che fa parte dell'interfaccia, ma che lasceremo vuoto, ed il metodo *onSensorChanged()* che verrà chiamato dal sistema ad ogni cambiamento dei dati del sensore e nel quale prenderemo le nostre decisioni.
In questo metodo, grazie alla precedente registrazione abbiamo anche l'informazione (sempre passata dal sistema) dell'evento scatenato dal sensore (*event*) dal quale possiamo ottenere i dati veri e propri (*event.values*) che nel nostro caso sono rappresentati da un array di float contenente il valore di accelerazione sui tre assi, ed anche l'indicazione del sensore (*event.sensor*) che ha scaturito l'evento.
L'obiettivo è quello di generare una chiamata verso un numero di telefono ogni qual volta il dispositivo venga scosso ripetutamente, quindi dobbiamo verificare che ci siano forti variazioni di accelerazione su tutti gli assi, in un tempo più o meno breve. Questo viene realizzato semplicemente memorizzando gli ultimi valori e sottraendoli a quelli correnti, per poi contare il numero di "scosse" fino a raggiungere un numero limite (*SHAKE_COUNT*) oltre il quale verificheremo che il tempo trascorso dal primo scuotimento non superi un certo valore massimo (*SHAKE_DURATION*) per decidere se effettivamente il dispositivo è stato scosso sufficientemente. Per gestire la variabile "tempo" abbiamo usato la funzione di sistema *System.currentTimeMillis()* che restituisce i millisecondi trascorsi dal gennaio 1970.
Se tutte queste soglie vengono superate con successo, attiveremo la chiamata vera e propria al numero di telefono desiderato tramite la funzione *callNumber()* che vedremo adesso.

FUNZIONI GSM

Con Android possiamo gestire chiamate e messaggi GSM anche all'interno della singola applicazione; noi utilizzeremo questa possibilità per inviare comandi ad un *GSM Shield* montato su Arduino, che sulla base dei messaggi ricevuti sarà in grado di attivare apparati esterni o risponderci a sua volta con invio di SMS contenenti particolari informazioni.
Per gestire una chiamata in uscita abbiamo cre-

Listato 3

```
private void callNumber(String numberToCall)
{
	try
	{
		Intent callIntent = new Intent(Intent.ACTION_CALL);
		callIntent.setData(Uri.parse("tel:" + numberToCall));
		startActivity(callIntent);

		Log.v(TAG,"GSM Call Number:" + numberToCall);
	}
	catch (ActivityNotFoundException e)
	{
		Log.e("GSM Call Example", "Call failed", e);
	}
}



private void sendMessage(String phoneNumber, String message)
{
	SmsManager smsManager =  SmsManager.getDefault();
	smsManager.sendTextMessage(phoneNumber, null, message, null, null);
}
```

Listato 4

```
private PhoneStateListener _phoneListener = new PhoneStateListener()
{
	public void onCallStateChanged(int state, String incomingNumber)
	{
try
		{
			switch (state)
			{
				case TelephonyManager.CALL_STATE_RINGING:
					Toast.makeText(GSMControllerActivity.this, "CALL_STATE_RINGING
				FROM " + incomingNumber, Toast.LENGTH_SHORT).show();
				break;

case TelephonyManager.CALL_STATE_OFFHOOK:
					Toast.makeText(GSMControllerActivity.this,
					"CALL_STATE_OFFHOOK", Toast.LENGTH_SHORT).show();
				break;

case TelephonyManager.CALL_STATE_IDLE:
					Toast.makeText(GSMControllerActivity.this, "CALL_STATE_IDLE",
				Toast.LENGTH_SHORT).show();
				break;

				default:
					Toast.makeText(GSMControllerActivity.this, "default",
				Toast.LENGTH_SHORT).show();

Log.i("Default", "Unknown phone state=" + state);
				}
			}
			catch (Exception e)
			{
			  Log.i("Exception", "PhoneStateListener() e = " + e);
			}
	}
};
```

ato la funzione *callNumber()* nella quale, come si vede dal **Listato 3**, utilizza un Intent di sistema (*ACTION_CALL*) che, dopo essere stato configurato col numero di telefono desiderato, viene passato alla funzione *startActivity()* in seguito alla quale ci troveremo l'interfaccia grafica che siamo abituati a vedere dopo aver composto un numero sul nostro telefono Android.

Al termine della chiamata (il dispositivo remoto sarà impostato, tramite Arduino, a concludere la chiamata e ad effettuare l'azione desiderata) verrà nuovamente visualizzata l'interfaccia della nostra Activity.
Come al solito, non dobbiamo dimenticarci di inserire nel file AndroidManifest.*xml*, tra i vari permessi, la seguente riga:

Listato 5

```
#include <GSM_Shield.h>
GSM gsm;

void setup()
{
  gsm.TurnOn(9600);
  gsm.InitParam(PARAM_SET_1);
  gsm.Echo(1);

  pinMode(13, OUTPUT);
  pinMode(11, OUTPUT);
}


void loop()
{

    int call;
    call=gsm.CallStatus();
    switch (call)
          {
      case CALL_NONE:
        Serial.println("no call");
        break;
      case CALL_INCOM_VOICE:
        delay(5000);
        gsm.PickUp();
        break;
      case CALL_ACTIVE_VOICE:
        delay(5000);
        gsm.HangUp();

          // do something
          digitalWrite(13, HIGH);

          break;
    }

    String sms = getSMS();

          if (sms == "cmd1")
          {
                    digitalWrite(13, LOW);
          }

          if (sms == "cmd2")
          {
                    digitalWrite(11, LOW);
          }

          delay(1000);


}

String getSMS()
 {
     char indexSms;
     indexSms =gsm.IsSMSPresent(type_sms);

     char numberCalling[15];
          char smsBody[122];

          String smsStr = "";

     if (indexSms!=0)
     {

       gsm.GetSMS(indexSms,numberCalling,
smsBody,120);
            smsStr = smsBody;
          }

     return smsStr;
 }
```

<uses-permission android:name="android.permission.CALL_PHONE"/>
Per quanto riguarda l'invio di un messaggio, nulla ci vieta di usare anche in questo caso l'Activity di sistema con cui siamo soliti inviare messaggi, ma c'è anche l'opportunità di ottenere lo stesso risultato usando la classe *SmsManager*, senza visualizzare ulteriori interfacce. Questo è possibile, sempre nel **Listato 3**, attraverso la funzione *sendMessage()* in cui la stringa *message* rappresenta il corpo del messaggio e *phoneNumber()* il numero di destinazione.
Nella nostra applicazione assoceremo l'invio di particolari messaggi preconfigurabili a tre pulsanti che inseriremo con la procedura ormai standard nell'Activity principale. Anche in questo caso non dovremo dimenticarci di inserire il seguente permesso:
<uses-permission android:name="android.permission.SEND_SMS" />
nel file *AndroidManifest.xml* e di aggiungere agli import precedenti il seguente:
import android.telephony.SmsManager;
Anche per la gestione delle chiamate in entrata, l'approccio è asincrono, ma ormai sappiamo usare le classi Listener viste prima per il caso dell'accelerometro e quindi non dovrebbero esserci problemi. Useremo la classe *TelephonyManager* e pertanto inseriremo il relativo import:
import android.telephony.TelephonyManager;
e il permesso nel *ManifestAndroid.xml*:
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
A questo punto nella *onCreate()* del nostro servizio, attraverso la classe *TelephonyManager*, registreremo la classe *PhoneStateListener* con le seguenti righe di codice:
TelephonyManager _telephonyManager=TelephonyManager) getSystemService(TELEPHONY_SERVICE);
_telephonyManager.listen(_phoneListener, PhoneStateListener.LISTEN_CALL_STATE);
Ovviamente dobbiamo prima implementare la classe *_phoneLiestener* come nel **Listato 4**, in cui andremo a riempire il metodo *onCallStateChanged()* che viene chiamato dal sistema operativo ogni volta che viene cambiato lo stato della chiamata.
Sapendo in automatico anche il numero chiamante (*incomingNumber*) potremmo controllarlo ed eseguire le opportune funzioni. Per ora, siccome quella proposta è un'applicazione di-

Listato 6

```
private LocationListener _locationListener = new LocationListener()
{
        public void onLocationChanged(Location location)
        {
                _vibrate.vibrate(300);

                if (location == null)
                {
                        return;
                }

                _lastLocation = location;
                _buttonStoreLocation.setEnabled(true);
                _buttonStoreLocation.setBackgroundColor(Color.GREEN);

                float speed = location.getSpeed();
                double altitude = location.getAltitude();
                double longitude = location.getLongitude();
                double latitude = location.getLatitude();

                float distanceMeters = (float)0;
                if (_storedLocation != null)
                {
                        distanceMeters = location.distanceTo(_storedLocation);
                        if(distanceMeters < _radiusGPS)
                        {
                                // do somethings
                        }
                }

                String dbgInfoString = "LAT:"+ latitude + "\n" + " LON:" +longitude + "\n" + " ALT:" +
        altitude + "\n" + " SPD:" + speed + "\n" + "DST:" + distanceMeters;
                Log.v(TAG, dbgInfoString);
                Toast.makeText(GSMControllerActivity.this, dbgInfoString , Toast.LENGTH_LONG).show();

                NumberFormat numberFormat = new DecimalFormat("0.000");
                _textCurrentLocation.setText("LAT:"+ numberFormat.format(latitude) + "    LON:" +
        numberFormat.format(longitude));


        }

        public void onStatusChanged(String provider, int status, Bundle extras)
        {
                Toast.makeText(GSMControllerActivity.this, status , Toast.LENGTH_SHORT).show();
        }
};
```

dattica, ci siamo limitati a visualizzare messaggi Toast.
Notate che i messaggi Toast sono "permessi" dal servizio, seppure rappresentino una sorta di interfaccia grafica con l'utente. Questo vuol dire che se anche la nostra Activity non è in primo piano, ma il nostro servizio è attivo, quando riceveremo una chiamata, indipendentemente dallo stato del nostro dispositivo riceveremo sullo schermo la notifica grafica tramite il messaggio Toast.
Adesso diamo una rapida occhiata al lato Arduino che riceverà le chiamate ed attiverà le uscite verso altri dispositivi ad esso collegati.
Nel **Listato 5** possiamo vedere un esempio di come usare la libreria *GSM_Shield* su Arduino, per gestire il controllo delle chiamate e degli SMS in ingresso. Periodicamente controlleremo le chiamate in arrivo e verificheremo la presenza di SMS in arrivo per abilitare o meno due piedini di uscita a seconda del messaggio ricevuto.

UTILIZZO DEL GPS

La nostra applicazione ha ora una sua praticità, ma possiamo ancora dotarla di ulteriori caratteristiche, come far effettuare una chiamata o inviare un SMS verso lo shield GSM ogni qualvolta lo smartphone o tablet si trovi nell'area intorno ad un punto geografico.
Useremo, allo scopo, il GPS, che non è un sensore della lista accennata in precedenza ma costituisce un modulo a parte; il meccanismo con il quale possiamo venire in possesso dei dati (latitudine e longitudine) corrispondenti alla posizione in cui si trova il dispositivo, è tuttavia il medesimo: esiste un *manager* (*LocationManager*) che si occupa esclusivamente di gestire la posizione e che viene utilizzato per registrare una classe Listener

Listato 7

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android" >
 <PreferenceCategory
   android:title="Gsm numbers information"
   android:summary="Destination phone number" >
  <EditTextPreference
     android:key="numberToCallAfterShaking"
     android:title="Phone Number to Call after shaking"
     android:summary="Called after shaking your device" />
  <EditTextPreference
     android:key="numberToCallInLocation"
     android:title="Phone Number to Call in location X"
    android:summary="Called when your device is in stored location" />
 </PreferenceCategory>

 <PreferenceCategory
   android:title="Accelerometer Settings"
   android:summary="Set the shaking sensibility" >

  <CheckBoxPreference
     android:key="chbShaking"
     android:title="Call Shaking Enable"
     android:summary="On/Off"
     android:defaultValue="true" />

   <EditTextPreference
     android:key="shakeDuration"
     android:title="Shake Duration [ms]"
     android:summary="start call after this time"
     android:defaultValue="600"
    android:dependency="chbShaking" />

   <EditTextPreference
     android:key="shakeCount"
     android:title="Shake Count"
     android:summary="start call after count shake"
     android:defaultValue="4"
          android:dependency="chbShaking" />

 </PreferenceCategory>

 <PreferenceCategory
   android:title="Location Settings"
   android:summary="Set the calling in location" >

  <CheckBoxPreference
     android:key="chbLocation"
     android:title="Call Location Enable"
     android:summary="On/Off"
     android:defaultValue="true" />

  <ListPreference
     android:key="listProviderLocation"
     android:title="Location Provider"
     android:summary="Choice GPS or Network"
     android:entries="@array/listOptions"
     android:entryValues="@array/listValues" />

 </PreferenceCategory>

</PreferenceScreen>
```

(*LocationListener*) in diverse modalità. Questo è presentato nelle seguenti righe, che vanno scritte nel metodo *onStart()* del nostro servizio:

```
Location Manager _locationManager = (LocationManager) this.getSystemService(LOCATION_SERVICE);
_locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, _locationListener);
```

È possibile determinare la posizione anche tramite Internet (*NETWORK_PROVIDER*) anche se con una precisione minore. Ovviamente è necessario che il nostro smartphone sia impostato per utilizzare rete Wi-Fi o GPS, cosa che potremmo sempre verificare da codice ed eventualmente utilizzare un Intent di sistema per visualizzare direttamente la configurazione Android con cui settare i metodi da utilizzare, attraverso queste due semplice linee di codice:

```
Intent intent = new Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS);
startActivity(intent);
```

Il risultato dell'operazione è visibile in **Fig. 6**.
La corrispondente classe Listener è rappresentata nel **Listato 6**; ad ogni cambiamento di posizione viene chiamata la *onLocationChanged()* la quale ci fornisce l'oggetto *Location* contenente la posizione espressa con longitudine, latitudine e, se il numero di satelliti "agganciati" lo permette, anche altitudine. Oltre a questo, tramite Location siamo anche in grado di calcolare la distanza in metri da un punto desiderato (grazie al metodo *distanceTo()*) che avremo settato in precedenza premendo il pulsante *_storedLocation*.
A questo punto, per far scattare la stessa chiamata di prima o l'invio di particolari messaggi basterà controllare quando questa distanza diventi minore di un certo raggio, dato che ora abbiamo tutto il necessario per farlo.
Gli import per gestire il GPS sono i seguenti:

```
import android.location.LocationManager;
import android.location.LocationListener;
import android.location.Location;
```

I permessi nel file *ManifestAndroid.xml* sono come in **Fig. 5**. In particolare, *ACCESS_FINE_LOCATION* è utilizzato se usiamo il GPS, mentre *ACCESS_COARSE_LOCATION* serve se usiamo la rete. Possiamo comunque metterli

Fig. 6

PASSAGGIO PARAMETRI TRA ACTIVITY E SERVIZI

Dovendo spesso gestire applicazioni con più di una Activity e servizi, occorre trasferire alcuni parametri tra le varie classi, non solo tra un'Activity ed uno o più servizi, ma anche tra più Activity. Esistono diversi modi più o meno eleganti; ne considereremo due oltre a quello più complesso di creare una finestra di opzioni (Preferences) che rimangono condivise in tutta l'applicazione come abbiamo analizzato in queste pagine.
Il primo è quello di fornire all'Intent informazioni aggiuntive e personalizzate, dette appunto "extra" che verranno lette dal servizio o dall'Activity di destinazione, dopo aver chiamato la *startActivity()* o la *startService()* dalla classe principale. Per scrivere e leggere queste informazioni "extra" utilizzeremo due metodi dell'Intent, rispettivamente *putExtra()* e *getExtra()*.
L'esempio in **Fig. A** chiarisce ciò.

ACTIVITY

Fig. A

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

    _serviceIntent = new Intent(getApplicationContext(), GSMService.class);

    _buttonStart = (Button) findViewById(R.id.buttonStart);
    _buttonStart.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v)
        {
            _serviceIntent.putExtra("Latitude", _latitude );
            _serviceIntent.putExtra("Longitude", _longitude );
            startService(_serviceIntent);

            _textServiceStatusInfo.setText("Service Status: Enabled");
        }
    });
```

SERVICE

```
@Override
public void onStart(Intent intent, int startId)
{
    Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
    Log.d(TAG, "onStart");

    _statusService = true;

    Bundle bundle = intent.getExtras();
    double lat    = bundle.getDouble("Latitude");
    double lon    = bundle.getDouble("Longitude");

    Log.d(TAG, "Received values:" + lat + " " + lon);

}
```

Le stringhe "*Latitude*" e "*Longitude*" rappresentano le chiavi, mentre i valori veri e propri (in questo caso di tipo double...) sono *_latitude* e *_longitude*.

```
@Override
public void onDestroy()
{

    //...
    // save the preferences
    SharedPreferences prefs = getSharedPreferences(MYPREFS, Context.MODE_PRIVATE);
    SharedPreferences.Editor editor = prefs.edit();
    editor.putString("PHONE_NUMBER", newPhoneNumber);
    editor.putInt("RADIUS", newRadius);
    editor.commit();

}
```

```
@Override
public void onCreate()
{
    // load the preferences
    SharedPreferences prefs = getSharedPreferences(MYPREFS, Context.MODE_PRIVATE);
    String phonenumber = prefs.getString("PHONE_NUMBER", "No Phone");
    int radius         = prefs.getInt("RADIUS", 0);

    //...
}
```

Fig. B

Nell'esempio, il destinatario è il servizio che ha già l'Intent come parametro, ma se fosse stata un' altra Activity occorre ricavare l'Intent associato tramite la funzione *getIntent()*. Inoltre l'oggetto *Bundle*, ha naturalmente, oltre a *getDouble()*, tutti gli altri metodi per ritornare i tipi corrispondenti. Il secondo metodo si basa sulla classe **SharedPreferences** ed usa lo stesso meccanismo delle Preferences, senza però dover creare un' interfaccia grafica di settaggi. Potremmo un po' paragonarlo al *registry* di Windows, perché si basa sull'associazione *chiave-valore* e perché rimane persistente nel sistema anche dopo la chiusura dell'applicazione. Queste informazioni vengono salvate in un file all'interno del percorso *data/data/* e possono essere condivise anche da altre applicazioni.
Nell'esempio di **Fig. B** sono state usate all'interno della stessa Activity per memorizzare informazioni e recuperarle al successivo riavvio, ma nulla vieta di usarle anche tra Activity o servizi diversi, in qualsiasi punto del codice.
La stringa *MYPREFS* identifica l'associazione tra le due sezioni e può contenere qualsiasi testo, basta che sia univoca. La costante *MODE_PRIVATE* indica, invece, che le informazioni saranno valide solo per la nostra applicazione.
Infine al metodo *getString()* deve essere passato come parametro, oltre alla chiave, anche il valore predefinito nel caso in cui non venga trovata nessuna chiave corrispondente.

entrambi, qualora prevedessimo di poter gestirli nella stessa applicazione.

PREFERENCES ANDROID

L'applicazione sarebbe conclusa, ma per renderla più versatile inseriremo una serie di opzioni con le quali gestire alcuni parametri come il raggio d'azione della localizzazione, l'entità dello scuotimento superata la quale viene effettuata la chiamata, i numeri destinatari, il corpo dei messaggi che verranno inviati sulla pressione dei pulsanti e tante altre. Android ci viene incontro fornendoci gli strumenti per creare, con semplicità, una schermata di opzioni raggruppabili con le relative intestazioni. Questa schermata, o meglio Activity, è detta **Preferences** ed è la stessa usata da Android quando accediamo alle impostazioni di sistema (o *Settings*). Viene usata generalmente per gestire grosse quantità di opzioni ed offre i vantaggi di condividere le informazioni scelte dall'utente anche a tutte le altre Activity del progetto. In questo modo le modifiche che faremo nelle Preferences (avviate dall'Activity principale) saranno accessibili anche all'interno del nostro servizio, dove avranno il loro effetto. Inoltre le modifiche che effettuere-

Fig. 7

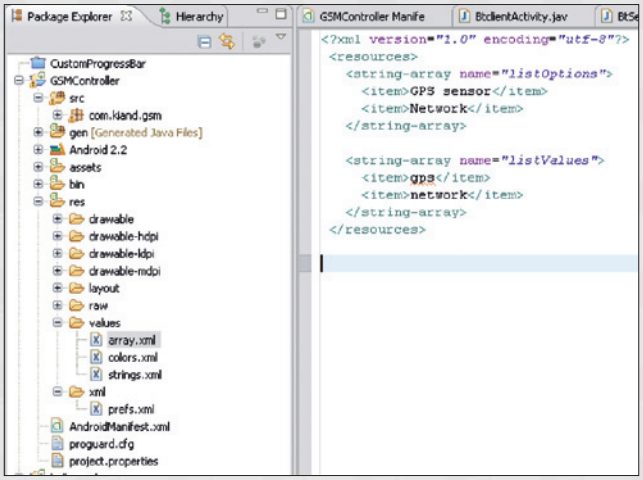


Fig. 8

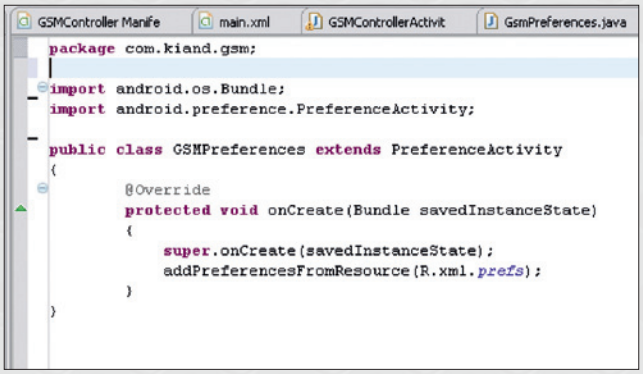
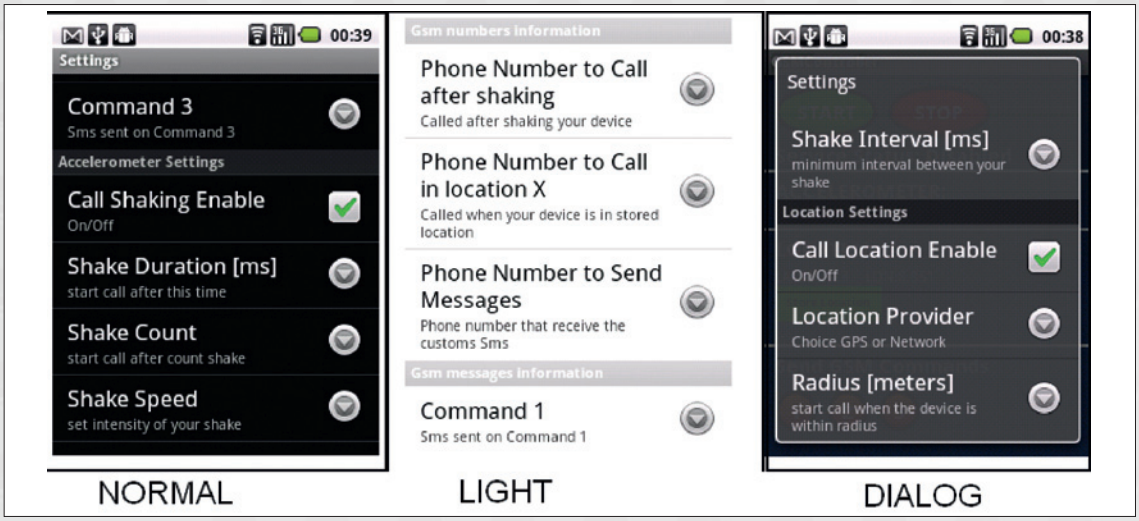


Fig. 9

mo all'interno delle Preferences verranno salvate automaticamente nel sistema e le ritroveremo uguali anche al successivo riavvio dell'applicazione senza bisogno di alcun intervento da parte nostra.
Vediamo ora come creare una nostra finestra di Preferences a partire da quello che rappresenterà la nostra interfaccia grafica: il file xml (nel nostro caso *prefs.xml*) che andremo a creare all'interno della cartella *xml* interna alla cartella di sistema *res*. Se questa non fosse già presente, potremo crearla da browser ed eseguire un "refresh" del progetto. Un frammento (non integrale) di questo file è visibile nel **Listato 7**. Come si può notare, la filosofia è la stessa dei file di layout, ma questa volta non dobbiamo preoccuparci delle dimensioni e della posizione degli oggetti grafici, i quali potranno essere principalmente di tre tipi: **CheckBoxPreference** (usata per valori booleani), **EditTextPreference** (usata per le stringhe) e **ListPreference** (usata per scegliere valori all'interno di una lista).
Ognuno di questi oggetti viene rappresentato dall'intestazione principale (*android:title*), da una descrizione che comparirà sotto il titolo (*android:summary*) e da una chiave (*android:key*) che deve essere univoca per ogni controllo, oltre ad altri campi opzionali come ad esempio *android:defaultValue* per impostare valore di default ed *android: dependency* che verrà impostato con la chiave di un controllo CheckBoxPreference, ad indicare che l'oggetto grafico sarà abilitato o meno a seconda del valore del checkbox indicato.
Ognuno di questi oggetti potrà anche essere raggruppato all'interno di una categoria mediante il tag xml **PreferenceCategory**, anch'esso dotato del campo *title* e *summary*.
Per quanto riguarda l'oggetto ListPreference, esso merita un discorso a parte, perché avrà un campo per contenere gli elementi della lista (*android:entries*) ed un altro (*android:entryvalues*) che contiene i valori restituiti in seguito alla scelta dell'elemento corrispondente. Solitamente si sceglie di elencare questi valori in un file xml a parte (nel nostro caso *array.xml*) che conterrà questi array e che andremo ad aggiungere nella cartella *values* all'interno della cartella *res*, come in **Fig. 7**. Prima di scendere nei dettagli possiamo vedere in **Fig. 10** un'anticipazione di come si presenterà.
Ora è il momento di associare l'interfaccia xml ad una nuova Activity, quindi andremo a creare un nuovo file ed una nuova classe che chiameremo *GSMPreferences* e che sarà una classe estesa della classe *PreferenceActivity*. Nel metodo *onCreate()* di **Fig. 8** useremo la funzione *addPreferencesFromResources()* cui passeremo il nostro file xml precedentemente creato (*prefs.xml*); da notare l'import *PreferenceActivity* necessario per la compilazione della classe.
Questa classe, come vedete, è molto corta perché contiene solamente l'associazione con il file xml, ma sempre nella *onCreate()* potremmo aggiungere il codice per accedere agli oggetti grafici

Fig. 10



delle Preferences ed usarlo per controllare, ad esempio, la consistenza dei valori inseriti.
Anziché usare il classico *findViewById()* al quale eravamo abituati con le Activity, basterà usare il metodo *findPreference()* cui passeremo la chiave (*android:key*) dell'oggetto. Ottenuto il controllo potremmo usare allo stesso modo, una classe Listener (*setOnPreferenceChangeListener*) che implementeremo a piacere e che verrà chiamata dal sistema non appena quel controllo verrà impostato ad un certo valore.
Adesso dobbiamo solo creare un pulsante o una voce di menu (già visto nella puntata precedente) da cui lanciare le nostre Preferences e per farlo useremo il solito modo utilizzato per le Activity normali, creando una nuova Intent col nome della classe (*GSMPreferences.class*) ed usando la *StartActivity()* come nelle righe seguenti:

```
Intent iPrefs = new Intent(this, GSMPreferences.class);
startActivity(iPrefs);
```

Essendo le Preferences un'Activity in tutto e per tutto, non dobbiamo dimenticarci di aggiungerle al solito file *AndroidManifest.xml* come in figura **Fig. 9**.

Potremo infine personalizzare anche l'aspetto grafico delle Preferences a piacere, utilizzando il campo *android:theme* all'interno dell'Activity, come ad esempio:

```
android:theme="@android:style/Theme.Dialog"
```

e:

```
android:theme="@android:style/Theme.Light.NoTitleBar.Fullscreen"
```

Le due hanno diversi effetti, come si può vedere nella **Fig. 10**.
In ultimo vediamo come sia semplice accedere ai valori modificati nelle Preferences, da qualsiasi Activity o servizio. Basterà, infatti, un oggetto della classe **SharedPreferences** ed usare i suoi metodi opportuni per ricavare i numeri , le stringhe o i booleani dai controlli associati semplicemente utilizzando le chiavi che abbiamo inserito nel campo *android:key* del file *prefs.xml*. Un esempio è mostrato nel **Listato 8**, di cui la funzione *showUserSettings()* può essere chiamata in ogni file del progetto.

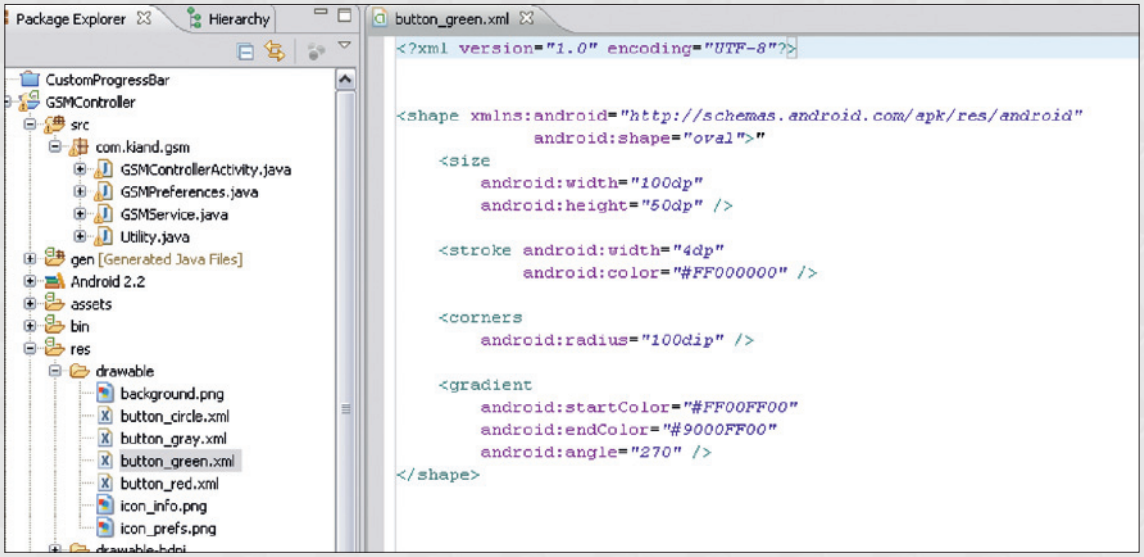
PULSANTI PERSONALIZZATI

Infine, per personalizzare ulteriormente la nostra applicazione possiamo dedicarci allo stile dei pulsanti che attivano e arrestano il

Listato 8

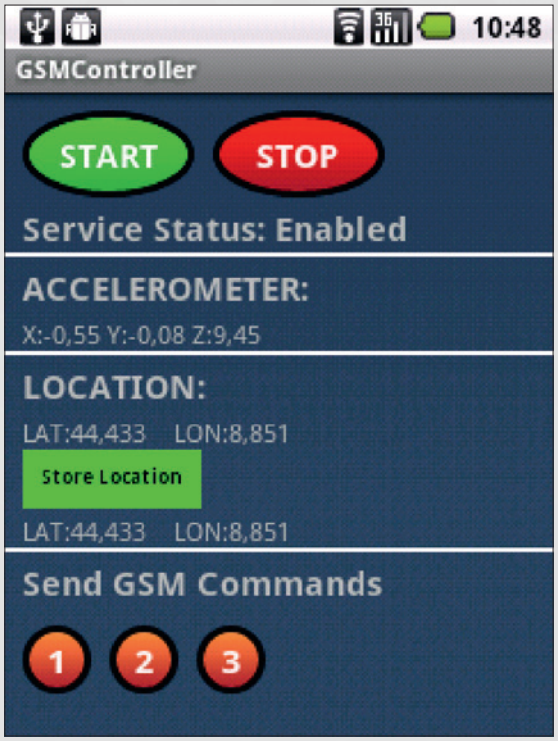
```
private void showUserSettings()
{
SharedPreferences sharedPrefs = PreferenceManager.getDefaultSharedPreferences(this);

          String numberAfterShaking = sharedPrefs.getString("numberToCallAfterShaking", "111111");
          String numberAfterLocation = sharedPrefs.getString("numberToCallInLocation",  "222222");
          boolean checkBoxShake = sharedPrefs.getBoolean("chbShaking", false);
          boolean checkBoxLocation = sharedPrefs.getBoolean("chbLocation", false);
          long shakeDuration = sharedPrefs.getLong("shakeDuration", 1000);
          int numberOfShakeCount = sharedPrefs.getInt("shakeCount",    4);
          int radius = sharedPrefs.getInt("radius", _defaultRadius);
}
```

Fig. 11



servizio. Abbiamo visto nella puntata precedente come cambiare il colore ad una SeekBar tramite aggiunta di un file di stile xml che andavamo ad aggiungere nella cartella *drawable*; se adesso vogliamo dare una forma particolare ai nostri pulsanti, possiamo seguire lo stesso procedimento andando a modificare il tag *shape* del nuovo file *button_green.xml* impostandolo al valore *oval*. Modificando anche i valori *width* e *height* del tag *size* possiamo decidere il grado di ovalizzazione o fare il pulsante precisamente rotondo. Potete trovare un esempio in **Fig. 11**.

Fig. 12



Nel nostro caso abbiamo creato tre file xml di pulsanti "nuovi" che renderemo attivi andando ad impostare il campo *android:background* nel tag *Button* del nostro file di Layout. Allo stesso modo della SeekBar potremmo anche creare gli effetti di ombra e colore per simulare la pressione del tasto. Il risultato finale è mostrato in **Fig. 12**.
Anche se l'accelerometro e il localizzatore GPS funzionano a livello di servizio, abbiamo aggiunto le stesse classi anche nell'Activity principale in modo da poter facilitare il debug.
Il tasto "Store Location" permette di memorizzare la posizione corrente che diventerà quella di riferimento e che attiverà la chiamata verso Arduino ogni qual volta ci troveremo nella stessa zona. Il passaggio di questo dato, dall'Activity al servizio avviene tramite il meccanismo dei metodi *putExtra()* e *getExtra()* associati all'Intent, come mostrato nel riquadro "Passaggio parametri tra Activity e Servizi" di queste pagine. Allo stesso modo avremmo potuto usare la classe SharedPreferences illustrata sempre nello stesso riquadro.

CONCLUSIONI

Ora abbiamo gli strumenti necessari per realizzare una nostra applicazione che risponda agli eventi esterni tramite i sensori analizzati e con la quale comunicare informazioni verso apparati elettronici esterni. Il tutto è reso anche più comodo grazie all'uso del servizio che non ci obbliga a tenere sotto controllo l'applicazione. Nella prossima puntata vedremo come creare un semplice Widget in modo da avere anche un feedback visivo sempre attivo sullo schermo del nostro dispositivo.

In questa puntata descriveremo la creazione di un Widget Android attraverso il quale potremo controllare e monitorare dispositivi remoti, collegati allo shield GSM Arduino, direttamente dallo schermo principale del nostro smartphone. Settima e ultima puntata.

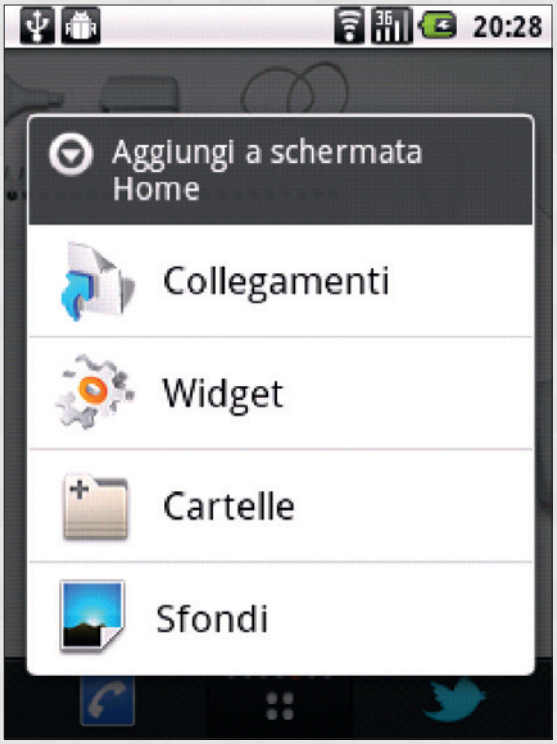
di Andrea Chiappori

7

PROGRAMMIAMO CON ANDROID

Abbiamo visto nello scorso articolo come implementare un servizio con il quale siamo stati in grado di attivare una chiamata telefonica o un messaggio SMS attraverso lo scuotimento del nostro dispositivo Android, o anche semplicemente attraversando una determinata area geografica. Un servizio ci permette di far funzionare la nostra applicazione in background senza dover tenerla in primo piano e continuando ad usare normalmente il nostro smartphone, ma non ci fornisce la possibilità di interagire con esso attraverso le solite interfacce grafiche come bottoni, liste e caselle di testo già esaminate in questo corso. Può, allora, risultare molto comodo l'utilizzo di un Widget, che altro non è che una vera e proprio applicazione Android visualizzata sullo schermo del cellulare sempre in primo piano (nella *homescreen*) e che potremo utilizzare per accedere in tempo reale alle informazioni del nostro apparato remoto. Solitamente ogni dispositivo Android, già con il firmware di base, offre una serie di Widget al proprio interno come ad esempio informazioni meteo, aggiornamenti sulle ultime notizie, riproduttori audio e tanti altri che permettono di avere sempre sotto controllo, in forma più semplificata ed immediata, i contenuti o le funzioni dell'applicazione a cui il Widget è legato. Per poter abilitare ed utilizzare questi Widget basta tenere premuto su una qualsiasi zona dello schermo principale fino alla comparsa di una finestra come quella in **Fig.1** in cui andremo a selezionare la voce Widget e successivamente a scegliere quello desiderato. Facilmente, una volta installata qualche applicazione, questa lista potrà aumentare perché spesso i Widget sono inclusi all'interno delle applicazioni e dopo l'installazione vengono resi disponibili al sistema. Tramite il Widget si ha spesso accesso

Fig. 1

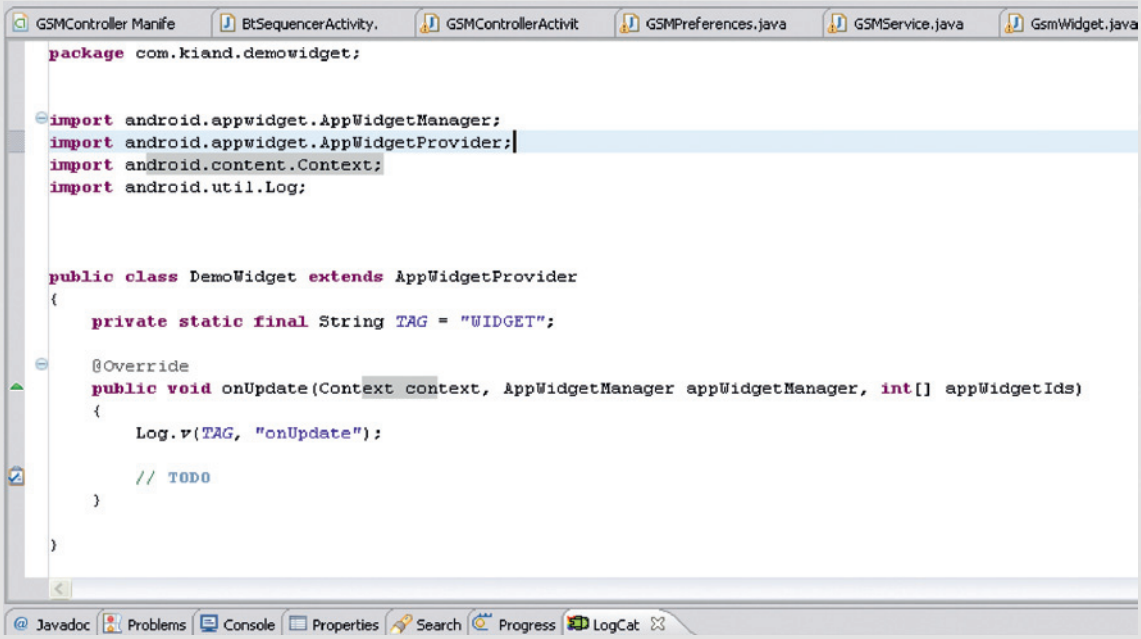


alle stesse informazioni e agli stessi comandi dell'applicazione vera e propria, ma con la comodità di poterlo fare direttamente in primo piano e sempre a portata di touch. Ovviamente nulla vieta di creare un Widget con funzionalità proprie, slegate da un'applicazione. In questo articolo vedremo come creare un primo Widget, che avrà una vita propria e che verrà aggiunto a quelli già presenti nel nostro dispositivo. Sarà un progetto dimostrativo senza particolare utilità, ma solo per facilitare la comprensione di questo elemento, mentre più interessante sarà vedere come dotare l'applicazione GSMController, vista nel numero scorso, di un suo Widget attraverso il quale inviare SMS e ottenere notizie a schermo sullo stato del servizio o su altre informazioni che possono risultare utili e offrire un valore aggiunto all'applicazione principale.

IL NOSTRO PRIMO WIDGET

Creiamo innanzitutto un nuovo progetto al solito modo, questa volta la nostra classe non sarà una classe estesa di un'Activity, ma della classe di sistema *AppWidgetProvider* della quale implementeremo inizialmente il metodo *onUpdate()* che sarà alla base del funzionamento del nostro Widget. Ricordiamo nuovamente che il nome del file .java dovrà essere lo stesso della nostra classe e quindi modificheremo il file *DemoWidget.java* come in **Fig. 2**, in cui è visibile la struttura base, anche se ancora vuota. Vedremo poi come andrà implementata, ma prima dobbiamo aggiornare il file *AndroidManifest.xml* come in **Fig. 3** in modo da indicare al sistema operativo il nome della classe che si occuperà di gestire il Widget e che è, appunto, *DemoWidget*.
Noterete come, non trattandosi di un'Activity, nel file *AndroidManifest.xml*, non compaia il tag *<activity>* ma un tag *<receiver>* nel quale è presente anche un'altra importante informazione sul nome del file xml che rappresenta la risorsa di informazione con la quale andremo

Fig. 2



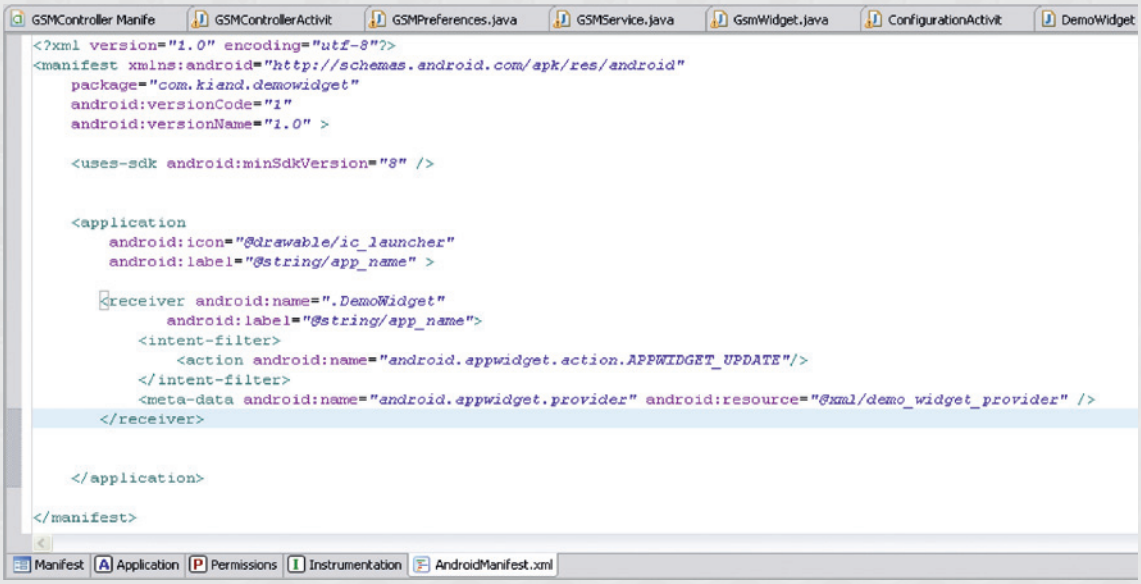


Fig. 3

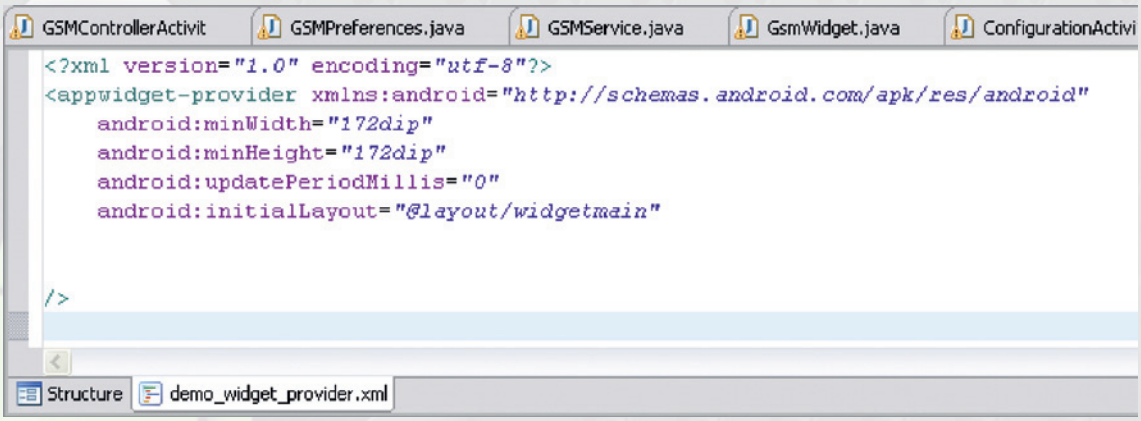
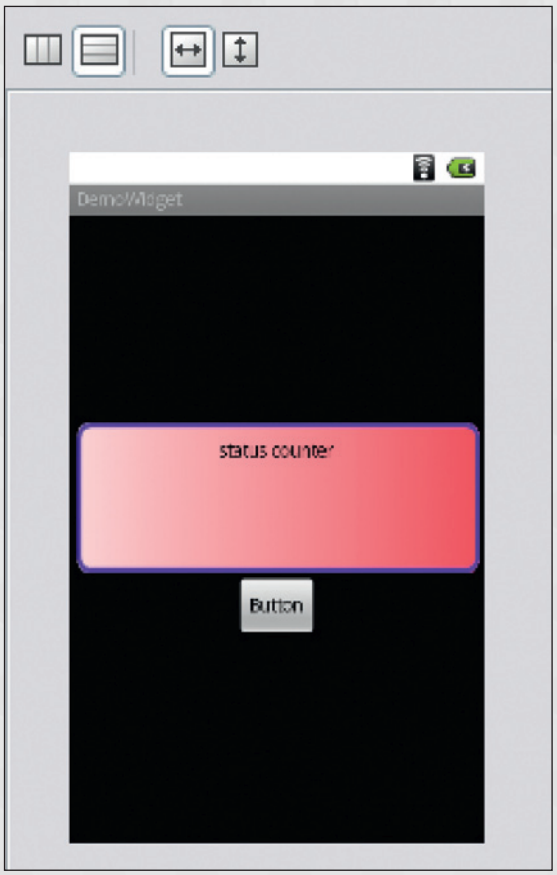


Fig. 4

ad impostare il nostro Widget. In questo caso si tratta del file *demo_widget_provider.xml* che è visibile in **Fig. 4** e tramite il quale siamo in grado di impostare alcune proprietà fondamentali come il periodo di refresh (*updatePeriodMillis*) con cui il sistema operativo chiama il metodo *onUpdate()* visto prima, le dimensioni minime con cui comparirà sullo schermo (*minWidth* e *minHeight*) ed il layout grafico (*initialLayout*) in cui andremo a disegnare l'interfaccia grafica personalizzata. In particolare, riguardo alle dimensioni minime, occorre cercare un compromesso evitando di utilizzare valori troppo piccoli che renderebbero il tutto illeggibile, ma nemmeno valori troppo grossi che ne impedirebbero la visualizzazione, specialmente nel caso in cui la *homescreen* fosse già occupata anche da altri Widget o da altre icone. Iniziamo ora a disegnare l'aspetto esteriore del nostro Widget creando il file di layout *widgetmain.xml* (indicato nel campo *initialLayout*) all'interno della cartella *res/xml*. Ricordiamo che nel caso in cui questa cartella non fosse già presente occorrerà crearla manualmente. Questo è un file di layout come quelli già visti e utilizzati nella creazione di una Activity ed è visibile in **Fig. 5**. Abbiamo usato un'immagine *png* per creare il riquadro colorato, una *TextView* per visualizzare una stringa (in particolare un semplice contatore) ed un controllo Button al quale assoceremo l'invio di una chiamata telefonica verso un numero di telefono preimpostato nel codice. A livello grafico non dobbiamo preoccuparci se l'immagine ed i controlli occupano tutta la parte dello schermo, perché al momento della visualizzazione sul dispositivo reale faranno fede i valori di altezza e larghezza impostati nel file di risorsa visto prima (*demo_widget_provider.xml*). A questo punto il progetto è consistente, possiamo passare a scrivere il cuore del nostro Widget implementando il metodo *onUpdate()* della classe *DemoWidget*, come nel **Listato 1**. Nel caso di un Widget, per accedere agli elementi grafici utilizzati nel layout viene usata la classe *RemoteView*, il cui costruttore riceve il percorso completo del package del progetto (nel nostro caso *com.kiand.demowidget*) ed il file di layout

Fig. 5



(*R.layout.widgetmain*); ora tramite il metodo *setTextViewText()* di questa classe saremo in grado di andare a scrivere una stringa nella TextView identificata da *id.widget_textview*. Per rendere attiva questa modifica occorre ancora effettuare un refresh su tutti i controlli tramite il metodo *updateAppWidget()* della classe *appWidgetmanager* che viene passata a *onUpdate()* e che riceve come parametri la classe del Widget e, appunto, le 'viste' (*remoteViews*) intese proprio come controlli. A questo punto, per testare il nostro Widget abbiamo creato una variabile interna alla classe, che incrementiamo ad ogni chiamata di *onUpdate()* da parte del sistema operativo e che stamperà la stringa sulla TextView. Purtroppo, però, a seconda della versione dell' ADK di Android che stiamo utilizzando potrebbe ancora non funzionare. Questo problema è dovuto al fatto che chiamare molte volte il metodo *onUpdate()* richiederebbe un consumo eccessivo della batteria anche quando il nostro dispositivo fosse in modalità *sleep*; la community Android suggerisce di utilizzare l'update non più di due volte all'ora e pertanto nelle prime versioni del sistema operativo il parametro *updatePeriodMillis* viene preso in considerazione solo per valori maggiori di 1800000 ms (appunto trenta minuti). Noi a scanso di equivoci utilizzeremo un metodo alternativo che prevede l'impiego della classe di sistema *Timer* e del suo metodo *scheduleAtFixedRate()*,

Listato 1

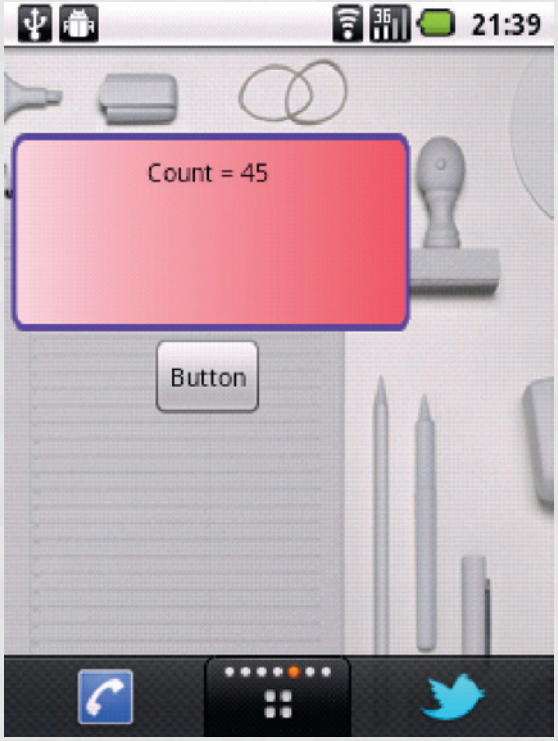
```
package com.kiand.demowidget;

import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.ComponentName;
import android.content.Context;
import android.util.Log;
import android.widget.RemoteViews;


public class DemoWidget extends AppWidgetProvider
{
	private static final String TAG = "WIDGET";

	private static int _counter = 0;

	@Override
	public void onUpdate(Context context, AppWidgetManager appWidgetManager, int[] appWidgetIds)
	{
		Log.v(TAG, "onUpdate");

		ComponentName thisWidget = new ComponentName(context, DemoWidget.class);
		RemoteViews remoteViews = new RemoteViews(context.getPackageName(), R.layout.widgetmain);

		remoteViews.setTextViewText(R.id.widget_textview, "TIME = " + _counter++);

		appWidgetManager.updateAppWidget(thisWidget, remoteViews);

	}
}
```

Listato 2

```
package com.kiand.demowidget;


import java.util.Timer;
import java.util.TimerTask;
import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.net.Uri;
import android.util.Log;
import android.widget.RemoteViews;
import android.app.PendingIntent;



public class DemoWidget extends AppWidgetProvider
{
	private static final String TAG = "WIDGET";


	private static Timer _timer = null;


	@Override
	public void onUpdate(Context context, AppWidgetManager appWidgetManager, int[] appWidgetIds)
	{
		Log.v(TAG, "onUpdate");


		_timer = new Timer();
		Counter count = new Counter(context, appWidgetManager);
		_timer.scheduleAtFixedRate(count, 1, 1000);


		RemoteViews remoteViews = new RemoteViews(context.getPackageName(), R.layout.widgetmain);

		// create intent for send a call
		Intent callIntent = new Intent(Intent.ACTION_CALL);
		callIntent.setData(Uri.parse("tel:" + "3331234567"));

		// link action to button1
		PendingIntent callPendingIntent = PendingIntent.getActivity(context, 0, callIntent, 0);
		remoteViews.setOnClickPendingIntent(R.id.button1, callPendingIntent);

		// update controls
		appWidgetManager.updateAppWidget(appWidgetIds, remoteViews);

	}

	@Override
    public void onDeleted(Context context, int[] appWidgetIds )
	{
		Log.v(TAG, "Deleted");

		if(_timer != null) {
			_timer.cancel();
			_timer.purge();
			   _timer = null;
			}
	}
```

come mostrato nel **Listato 2**, attraverso il quale siamo in grado di eseguire ad ogni intervallo di tempo ciò che implementeremo nel metodo *run()* di una nostra classe estesa di TimerTask e che abbiamo chiamato *Counter*. Basterà passare al metodo *scheduleAtFixedRate()* il nostro oggetto della nuova classe *Counter*, insieme al periodo di tempo espresso in millisecondi, ed aggiornare il controllo del Widget all'interno del metodo *run()* di questa classe. Ovviamente dovremo implementare il costruttore della classe *Counter* in modo da avere a disposizione tutti i dati necessari per accedere e modificare i controlli grafici del Widget che sono *Context* e *AppWidgetManager*. Sempre nel **Listato 2** noterete che sono stati implementati anche i metodi *onDeleted()* e *onDisabled()* della classe *DemoWidget* e che in essi è stato inserito il codice per la distruzione

Fig. 6



dell'oggetto *_timer*; questo è necessario per evitare che il codice contenuto nel metodo *onUpdate()* continui ad essere eseguito anche dopo aver disabilitato il Widget.
A questo punto possiamo eseguire il nostro primo Widget su emulatore o direttamente su dispositivo; il file eseguibile generato avrà sempre l'estensione *.apk*, ma questa volta, non trattandosi di una applicazione vera e propria, non verrà eseguito immediatamente dal sistema operativo, ma sarà semplicemente aggiunto alla lista dei Widget disponibili e potremo verificarne il funzionamento andando a selezionarlo nel solito modo. Il risultato che ci apparirà sarà come in **Fig. 6**. Al pari di ogni altro Widget, potrà essere trascinato a piacimento nelle zone dello schermo o anche spostato negli schermi laterali a disposizione.Per personalizzarlo maggiormente potremmo creare un'icona a nostro piacimento come già visto nelle puntate precedenti e la potremo vedere comparire accanto al nome del Widget nella lista interna di tutti gli altri Widget.

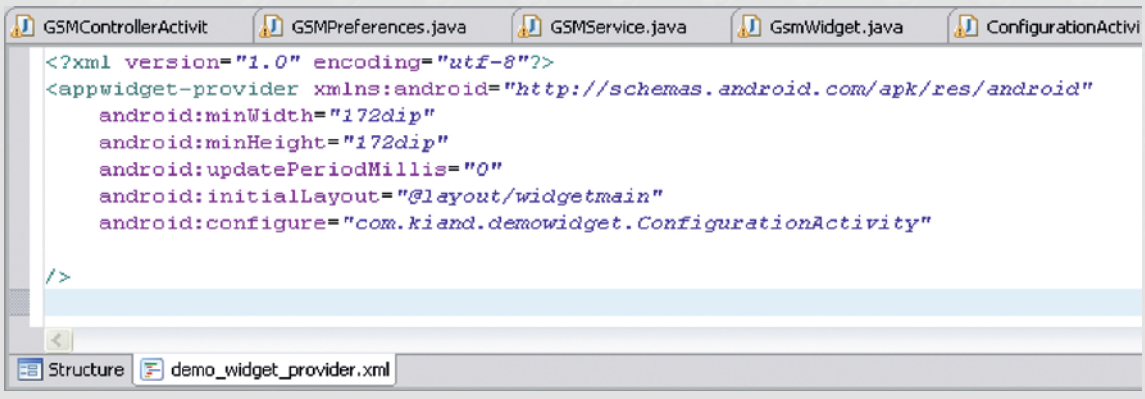
Listato 3

```
#include <GSM_Shield.h>
GSM gsm;



void setup()
{
  gsm.TurnOn(9600);
  gsm.InitParam(PARAM_SET_1);
  gsm.Echo(1);

  pinMode(13, OUTPUT);
  pinMode(11, OUTPUT);
}


void loop()
{

    int call;
    call=gsm.CallStatus();
    switch (call)
          {
      case CALL_NONE:
        Serial.println("no call");
        break;

      case CALL_INCOM_VOICE:
        delay(5000);
        gsm.PickUp();
        break;

      case CALL_ACTIVE_VOICE:
        delay(5000);
        gsm.HangUp();

                    digitalWrite(13, HIGH);

                    break;
    }

         delay(1000);
}
```

AGGIUNGIAMO I BOTTONI

Abbiamo, in realtà, già aggiunto un bottone a livello di layout grafico come risulta visibile in **Fig. 6**, ma non abbiamo ancora associato ad

Fig. 7



Listato 4

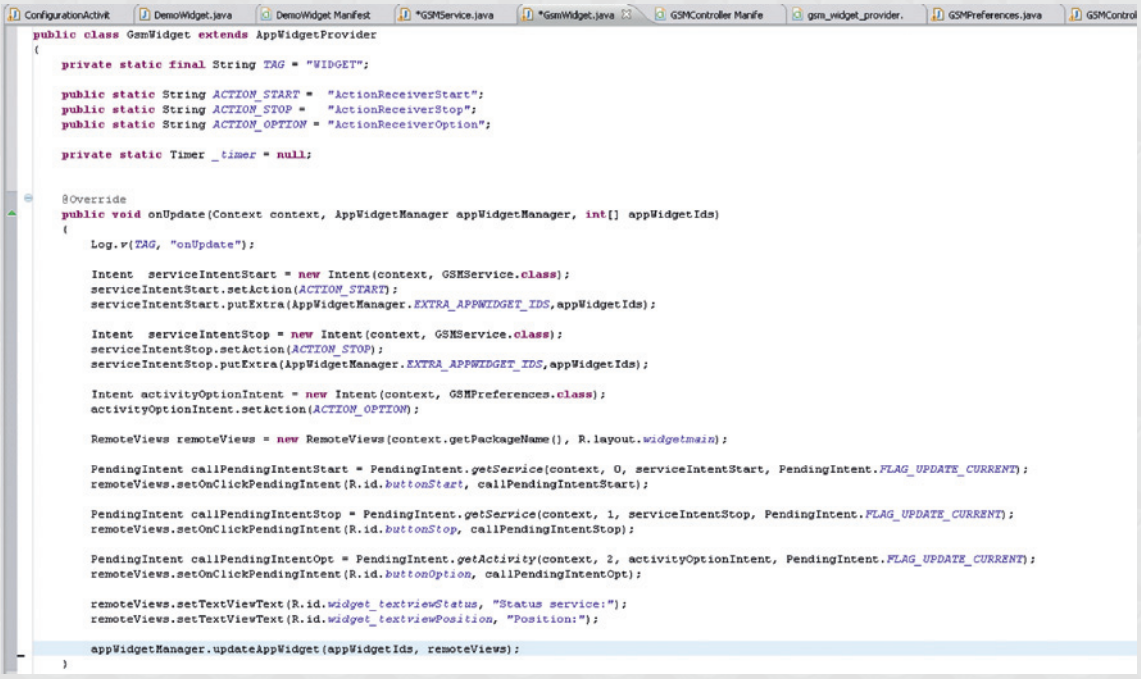
```
package com.kiand.demowidget;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;
import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.ComponentName;
import android.content.Context;
import android.util.Log;
import android.widget.RemoteViews;
import android.widget.Button;
import android.content.Intent;

public class ConfigurationActivity extends Activity
{
    private int widgetID;

    Button  _buttonOk;
    int mAppWidgetId = AppWidgetManager.INVALID_APPWIDGET_ID;

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

        widgetID = AppWidgetManager.INVALID_APPWIDGET_ID;
        Intent intent = getIntent();
        Bundle extras = intent.getExtras();
        if (extras != null)
        {
                widgetID = extras.getInt(AppWidgetManager.EXTRA_APPWIDGET_ID, AppWidgetManager.INVALID_APPWIDGET_ID);
        }

        _buttonOk = (Button)findViewById(R.id.buttonApply);
        _buttonOk.setOnClickListener(new View.OnClickListener()
        {
                @Override
                public void onClick(View v)
                {
                   Intent resultValue = new Intent();
                   resultValue.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, widgetID);
                   setResult(RESULT_OK, resultValue);
                   finish();
                }
        }
        );
```

esso l'implementazione del codice. Immaginiamo di voler eseguire una chiamata telefonica verso lo shield GSM di Arduino al quale sarà associata una determinata azione; per gestire la telefonata useremo lo stesso codice visto nella puntata precedente con cui creiamo l'Intent ACTION_CALL e dal quale creeremo un PendingIntent (pending perché usato in un altro processo) che passeremo al metodo *setOnClickPendingIntent()* della classe *RemoteView* vista prima e che rappresenta tutti i controlli del Widget. Assieme al *PendingIntent* passeremo anche l'ID della risorsa che è appunto il nostro bottone (*R.id.button1*). Tutto questo è visibile sempre nel **Listato 2**. Ora, senza dimenticarci di aggiungere gli import necessari e la riga di permission (*android.permission.CALL_PHONE*) nel solito file *AndroidManifest.xml* possiamo verificare il funzionamento scaricando nuovamente l'applicazione sul dispositivo. Ricordiamo che per cancellare il nostro Widget ci comporteremo come per qualsiasi altro Widget, tenendo premuto sul controllo per qualche secondo e trascinandolo sull' icona del cestino che comparirà sulla parte bassa (o alta) dello schermo. Anche se molto simile a quello della puntata precedentemente, possiamo dare un'occhiata al **Listato 3** nel quale è presentato il codice Arduino che gestisce la chiamata in arrivo attivando l'uscita del pin 13.

AGGIUNGIAMO LA CONFIGURAZIONE

Avrete notato come nel **Listato 2** il numero di telefono verso cui chiamare sia cablato nel codice; potremmo invece voler cambiare dinamicamente questo numero ogni volta che il Widget

Fig. 8



```
public class GsmWidget extends AppWidgetProvider
{
    private static final String TAG = "WIDGET";

    public static String ACTION_START =  "ActionReceiverStart";
    public static String ACTION_STOP =   "ActionReceiverStop";
    public static String ACTION_OPTION = "ActionReceiverOption";

    private static Timer _timer = null;


    @Override
    public void onUpdate(Context context, AppWidgetManager appWidgetManager, int[] appWidgetIds)
    {
        Log.v(TAG, "onUpdate");

        Intent  serviceIntentStart = new Intent(context, GSMService.class);
        serviceIntentStart.setAction(ACTION_START);
        serviceIntentStart.putExtra(AppWidgetManager.EXTRA_APPWIDGET_IDS,appWidgetIds);

        Intent  serviceIntentStop = new Intent(context, GSMService.class);
        serviceIntentStop.setAction(ACTION_STOP);
        serviceIntentStop.putExtra(AppWidgetManager.EXTRA_APPWIDGET_IDS,appWidgetIds);

        Intent activityOptionIntent = new Intent(context, GSMPreferences.class);
        activityOptionIntent.setAction(ACTION_OPTION);

        RemoteViews remoteViews = new RemoteViews(context.getPackageName(), R.layout.widgetmain);

        PendingIntent callPendingIntentStart = PendingIntent.getService(context, 0, serviceIntentStart, PendingIntent.FLAG_UPDATE_CURRENT);
        remoteViews.setOnClickPendingIntent(R.id.buttonStart, callPendingIntentStart);

        PendingIntent callPendingIntentStop = PendingIntent.getService(context, 1, serviceIntentStop, PendingIntent.FLAG_UPDATE_CURRENT);
        remoteViews.setOnClickPendingIntent(R.id.buttonStop, callPendingIntentStop);

        PendingIntent callPendingIntentOpt = PendingIntent.getActivity(context, 2, activityOptionIntent, PendingIntent.FLAG_UPDATE_CURRENT);
        remoteViews.setOnClickPendingIntent(R.id.buttonOption, callPendingIntentOpt);

        remoteViews.setTextViewText(R.id.widget_textviewStatus, "Status service:");
        remoteViews.setTextViewText(R.id.widget_textviewPosition, "Position:");

        appWidgetManager.updateAppWidget(appWidgetIds, remoteViews);
    }
```

viene avviato. Sappiamo ora come creare un altro bottone a cui associare un'Activity di configurazione, ma anche in questo caso Android ci viene incontro offrendoci un metodo più semplice: basterà aggiungere un nuovo campo, *android:configure*, nel file di risorsa *demo_widget_provider.xml* visto in precedenza e passargli il percorso del file java in cui scriveremo la nostra Activity di configurazione, come in **Fig. 7**. Nel nostro caso il file sarà *ConfigurationActivity.java* e conterrà una TextView in cui inserire il numero di telefono di destinazione ed un bottone con cui validarlo ed avviare il Widget.
Non sarà, in questo modo, necessario aggiungere un ulteriore pulsante al Widget per la sua configurazione perché la nostra *ConfigurationActivity* verrà automaticamente creata al momento del lancio del Widget e si occuperà lei stessa di

Fig. 9

```
    @Override
    public void onStart(Intent intent, int startid)
    {

        Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
        Log.d(TAG, "onStart Service");

        _statusService = true;


        // update  widget
        _appWidgetManager = AppWidgetManager.getInstance(this.getApplicationContext());
        _thisWidget = new ComponentName(getApplicationContext(),GsmWidget.class);
        _remoteViews = new RemoteViews(this.getApplicationContext().getPackageName(),R.layout.widgetmain);

        String strAction =  intent.getAction();
        if (strAction!= null)
        {
            if (strAction.equals(GsmWidget.ACTION_STOP))
            {
                _remoteViews.setTextViewText(R.id.widget_textviewStatus, "Status service: Stopped");
                _appWidgetManager.updateAppWidget(_thisWidget, _remoteViews);

                Log.d(TAG, "auto stop service");
                stopSelf();
                return;
            }
        }


        _remoteViews.setTextViewText(R.id.widget_textviewStatus, "Status service: Started");
        _appWidgetManager.updateAppWidget(_thisWidget, _remoteViews);

        //////////////
```

farlo partire, sulla pressione del bottone *buttonApply* come mostrato nel **Listato 4**.
Anche in questo caso dovremo aggiornare il file *AndroidManifest.xml* informandolo sull'utilizzo di questa nuova Activity e per passare l'informazione acquisita nella TextBox possiamo ricorrere alle *SharedPreferences* viste nella puntata precedente; il numero di telefono verrà memorizzato nell'Activity per poi essere recuperato ed utilizzato nel metodo *onUpdate()* del Widget stesso.

Listato 5

```
@Override
   public void onReceive(Context context, Intent intent)
   {
          String strAction = intent.getAction();
          Log.v(TAG, "onReceive:" + strAction);

       if (intent.getAction().equals(ACTION_START))
       {
          Log.v(TAG, "ACTION_START");
       }
       else if (intent.getAction().equals(ACTION_STOP))
       {
          Log.v(TAG, "ACTION_STOP");
       }
       else if (intent.getAction().equals(ACTION_OPTION))
       {
          Log.v(TAG, "ACTION_OPTION");
       } else {
           super.onReceive(context, intent);
       }
   }
```

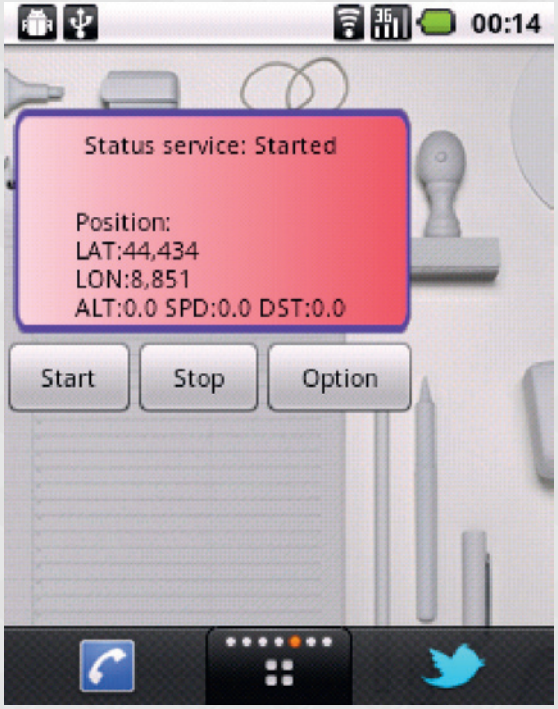
INTEGRAZIONE CON GSMCONTROLLER

Vediamo ora come aggiungere un Widget personalizzato alla nostra applicazione *GSMController*.
L'obiettivo è quello di aumentare visibilità al nostro servizio dotandolo di un controllo maggiore senza dover necessariamente riaprire ogni volta l'applicazione in questione. Possiamo ad esempio fare in modo che nel Widget vi sia un bottone per attivare direttamente il servizio, un altro per stopparlo e un terzo per accedere alle opzioni. Allo stesso modo possiamo fare in modo che il servizio dell'applicazione GSMController invii, esso stesso, determinate informazioni al Widget, come la distanza dal punto geografico prestabilito - aggiornata di volta in volta in tempo reale – o il numero di telefono corrente al quale verrà diretta la chiamata in caso di scuotimento ripetuto del dispositivo oltre allo stato del servizio ed altri dati utili.
Le modifiche da apportare sono relativamente poche, potremmo anche utilizzare parte del codice già scritto per il DemoWidget visto sopra, semplicemente copiando il file *DemoWidget.java* all'interno della cartella src del progetto, rinominandolo in *GsmWidget.java* e aggiungendo due bottoni al layout grafico (*widgetmain.xml*).
Per la visualizzazione delle informazioni potremo usare lo stesso controllo TextView utilizzato in precedenza, aggiungendone altri per visualizzare eventualmente altri dati con maggior dettaglio. Questa volta rinomineremo il file con le informazioni del Widget in *gsm_widget_provider.xml*, aggiorneremo opportunamente il file *AndroidManifest.xml* sulla base delle modifiche viste prima, ma senza utilizzare l'Activity di configurazione. Possiamo adesso compilare tutto e scaricare sul dispositivo l'applicazione che partirà normalmente, ma rispetto a prima, adesso abbiamo a disposizione un nuovo Widget che verrà identificato con la stessa icona dell'applicazione.
La parte più interessante da analizzare è invece l'interazione tra il nostro Widget e l'applicazione, perché vogliamo essere in grado di poterla controllare tramite il Widget in primo piano, ma anche di permettere all'applicazione stessa di aggiornare il Widget sul suo stato attuale.
Partiamo inizialmente a modificare il file *GsmWidget.java* ed in particolare il metodo *onUpdate()* che modificheremo come in **Fig. 8**.
Non utilizzeremo più la classe Timer per aggiornare periodicamente il Widget perché in questo caso non sarà necessario, anzi, basterà che il metodo *onUpdate()* sia chiamato una volta sola, al momento del lancio del Widget e per esserne

```
</activity>

<receiver android:name="com.kiand.gsm.GsmWidget"
      android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.appwidget.action.APPWIDGET_UPDATE"/>

            <action android:name="com.kiand.gsm.GsmWidget.ACTION_START"/>
            <action android:name="com.kiand.gsm.GsmWidget.ACTION_STOP"/>
            <action android:name="com.kiand.gsm.GsmWidget.ACTION_OPTION"/>


        </intent-filter>
        <meta-data android:name="android.appwidget.provider" android:resource="@xml/gsm_widget_provider" />
</receiver>
```

Fig. 10

Fig. 11



sicuri imposteremo il valore *updatePeriodMillis* a zero. Dovremo però associare ai tre bottoni le azioni desiderate e quindi creeremo due Intent per attivare e stoppare il servizio GSMController ed un Intent per chiamare l'Activity che gestirà le opzioni dell'applicazione principale (la PreferenceActivity *GsmPreferences*) già scritta ed analizzata nella puntata precedente.
Come già fatto per il Widget di esempio, ricaviamo i PendingIntent dai corrispondenti Intent appena creati e li associamo ai bottoni tramite il metodo *setOnClickPendingIntent()*, infine aggiorniamo le stringhe di testo nelle due TextView tramite *setTextVIewText()* e i controlli per rendere valide le modifiche con *updateAppWidget()*.
Da notare come tutti gli Intent vengano creati passando il nome della classe corrispondente (*GSMService* per il servizio e *GSMPreferences* per l'Activity) mentre per gli Intent che gestiscono lo start e lo stop del servizio si debba usare *PendingIntent.getService()* e per l'Activity occorra usare *PendingIntent.getActivty()*. Infine, non dimentichiamo di usare un requestcode (il secondo paramentro di *getService* e *getActivity*) diverso per la creazione dei vari PendingIntent, altrimenti tutti i bottoni avranno una stessa azione associata. Noterete anche come ogni Intent abbia associata una stringa diversa e personalizzabile, tramite il metodo *setAction()*; nel nostro caso sono ACTION_START, ACTION_STOP e ACTION_OPTION e servono per identificare l'Intent corrispondente. Ogni volta che viene premuto uno dei tre pulsanti, viene chiamato il metodo *onStart()* del servizio GSMService collegato all'Intent dal quale possiamo ricavare l'azione associata tramite il metodo *getAction()* ed essere in grado di capire quale pulsante sia stato premuto. Questo è quanto succede nella **Fig. 9** in cui ci troviamo già nel codice del servizio; in particolare se l'azione ricevuta è la ACTION_STOP interrompiamo il servizio nel quale già ci troviamo con la funzione di sistema *stopSelf()*. Nulla ci avrebbe vietato di usare il meccanismo di passaggio parametri dell'Intent tramite i metodi *putExtra()* e *getExtra()* visti nella scorsa puntata ed anzi, a seconda dei casi, si possono utilizzare entrambe le strade, ma l'impiego di queste Action ci offre anche il vantaggio di gestire la scelta dei pulsanti anche all'interno della classe Widget grazie al metodo *onReceive()* (nel file *GsmWIdget.java*) che è visibile nel **Listato 5** in cui possiamo implementare ulteriore codice di controllo. Per far sì che la *onReceive()* sia in grado di 'trappare' queste Action occorre registrarle nel file *AndroidManifest.xml* come mostrato in **Fig. 10**, all'interno del tag *<receiver>* e pertanto occorre che le Action siano dichiarate pubbliche nella classe del Widget.
Infine, una volta che nel servizio della nostra applicazione abbiamo ricavato l'oggetto *_remoteView* riferito ai controlli presenti nel Widget, come in **Fig. 9**, possiamo utilizzarlo per aggiornare il Widget con i dati che ci vengono forniti nel tempo dal servizio; ad esempio se volessimo visualizzare sulla TextView del Widget la locazione corrente o la distanza dal punto prefissato in cui partirà la chiamata verso lo Shield GSM di Arduino, potremmo farlo tranquillamente al solito modo, usando il metodo *setTextViewText()* seguito da *updateAppWidget()* ogni qual volta venga chiamata la *_locationListener* che è in attesa di una notifica dal sistema operativo di cambiamento di locazione. Il risultato finale sarà come in **Fig. 11**.

CONCLUSIONI

Con questa puntata concludiamo il corso di programmazione di Android, consapevoli di non aver approfondito alcuni argomenti di carattere prettamente informatico, talvolta senza scendere troppo nei dettagli, per prediligere il lato "hardware" dell'opera; speriamo vivamente di aver fornito comunque le principali basi e qualche spunto per i vostri futuri progetti elettronici nel mondo Android.