ESAME DI STATO A. S. 2015/2016

Emerjolly

Sara Tornincasa

Samuele Savoldi

I.T.I.S. "BENEDETTO CASTELLI"
CLASSE 5°AI INFORMATICA

INDICE

PRESENTAZIONE: *EMERJOLLY*

Il progetto da noi sviluppato è un'applicazione che, date delle parole chiave decise dall'utente, controlla ogni SMS ricevuto e, se queste sono presenti, fa suonare il cellulare indipendentemente dal fatto che sia in silenzioso, in vibrazione o in modalità suoneria.

Motivazioni

Il motivo per cui il telefono è stato creato è quello di poter comunicare a distanza: Antonio Meucci, inventore italiano accreditato da diverse fonti come inventore del primo telefono, nel 1854 realizzò un collegamento telefonico tra la camera di sua moglie gravemente malata e il suo laboratorio esterno, così da potersi tenere in contatto con lei.
L'evoluzione da telefono fisso a telefono cellulare è poi avvenuta di conseguenza: un mezzo trasmissivo portatile, in grado di funzionare ovunque senza bisogno di cavi rendeva ancora più semplice la comunicazione in caso di necessità.
Al giorno d'oggi i cellulari si sono evoluti in *smartphone*, termine entrato recentemente nel vocabolario comune. Uno smartphone è molto più di un telefono: offre un perenne collegamento a internet, la possibilità di scaricare qualsiasi tipo di applicazione, sia essa a scopo lavorativo o a scopo ludico, di ascoltare musica, scattare fotografie e registrare video. In tutto questo, la comunicazione viene per ultima.
La maggior parte delle persone ha il telefono in silenzioso o in vibrazione per la maggior parte della giornata, e non è raro che chi ha bisogno di parlare con noi aspetti molto tempo prima di riuscirci. Ma cosa succederebbe se ci fosse una vera emergenza, e noi fossimo a lavoro, a scuola, o fuori con gli amici, e non sentissimo le chiamate o i messaggi di un nostro genitore che ci chiede di andare al pronto soccorso perché il nonno sta male? La risposta è semplice: non ci andremmo, finché non prenderemo in mano il cellulare e ci accorgeremo dei tentativi di metterci in contatto con noi. Questo potrebbe accadere in tempo, ma potrebbe anche essere troppo tardi.
La nostra applicazione è pensata per le situazioni di emergenza, in modo da seguire il primo obiettivo della comunicazione telefonica: quello di chiedere aiuto, se necessario.
Una volta scaricata l'applicazione, alla prima apertura l'utente si troverà di fronte ad una schermata molto semplice: un bottone *switch* per renderla attiva e per disattivarla, una lista di parole chiave inizialmente riempita con parole di default, ed un pulsante per aggiungerne di nuove. E' inoltre possibile, tenendo premuta la parola scelta, accedere ad una schermata per modificarla e sostituirla con una nuova, oppure eliminarla direttamente. Le spiegazioni su come utilizzare l'applicazione vengono caricate al primo accesso nella lista, insieme alle parole di default. In questo modo l'utente può utilizzarle per fare delle "prove" e prendere confidenza con l'applicazione, eliminandole e modificandole.
Abbiamo deciso di svilupparla per la piattaforma Android, perché possediamo entrambi un telefono con questo sistema operativo.

ANDROID

Android è un sistema operativo per dispositivi mobili sviluppato da Google Inc. e basato su kernel Linux. È un sistema embedded, ovvero un sistema progettato per una determinata applicazione con una piattaforma hardware specifica che ne gestisce le funzionalità, progettato principalmente per smartphone e tablet. Esistono anche interfacce utente specializzate per televisori (Android TV), automobili (Android Auto), orologi da polso (Android Wear), occhiali (Google Glass) ed altri.
Il software maggiormente utilizzato per creare applicazioni per Android è Android Studio. Il codice viene scritto in linguaggio Java.

Un'applicazione è costituita da quattro blocchi fondamentali:

- Activity
- Intent
- Service
- Content Provider

Una volta decisi quali di questi componenti servono all'applicazione, bisogna elencarli in un file chiamato *AndroidManifest.xml*. Si tratta di un file XML in cui sono dichiarati tutti i permessi richiesti dall'applicazione, i suoi componenti, le relazioni tra di essi, le loro funzionalità e i requisiti.

ACTIVITY

Le Activity sono le più comuni tra i quattro blocchi che costituiscono le applicazioni Android. Un'Activity è normalmente una singola schermata nell'applicazione. Ogni Activity è implementata come una classe singola che estende la classe di base Activity. La classe presenterà un'interfaccia utente composta di Views, ovvero gli oggetti grafici, e risponderà agli eventi, come il click su un elemento. Il layout della grafica è definito in un file XML presente nella cartella *res > layout*.
La maggior parte delle applicazioni sono composte da schermate multiple: è possibile spostarsi tra di esse utilizzando gli Intent, altro blocco fondamentale di Android, che verrà spiegato in seguito.
In alcuni casi un'Activity può passare un valore all'Activity che chiama o che l'ha chiamata. Per esempio, un'Activity che permette all'utente di scegliere una foto restituirà la foto scelta a chi l'ha chiamata.

Un'Activity si divide in:

- Layout, ovvero la disposizione dei vari elementi grafici, specificata in un file XML autogenerato da Android Studio.
- Classe Java che estende la classe AppCompatActivity, e a cui viene assegnato il layout sopracitato. In questa classe sarà possibile gestire tutte le interazioni tra l'Activity e l'utente, come per esempio la pressione su un particolare bottone.

INTENT

Android usa una classe speciale chiamata Intent per la navigazione da schermata a schermata. Un Intent descrive cosa un'applicazione vuole che venga eseguito. Le due parti più importanti di un Intent sono l'azione da eseguire e i dati su cui agire.
Concretamente parlando, quando un Intent viene creato viene legato ad una Activity. Per proseguire nella navigazione viene utilizzato il metodo *startActivity(myIntent)*. La nuova Activity viene informata dell'Intent, causandone il lancio.
E' possibile utilizzare un IntentReceiver se si desidera che del codice nell'applicazione si esegua solo in reazione ad un evento esterno, per esempio quando suona il telefono, quando è disponibile la rete di dati, o quando scocca la mezzanotte. Questi non mostrano un'interfaccia utente, anche se possono usare il

NotificationManager per inviare delle notifiche.
L'applicazione non deve essere per forza in esecuzione perché vengano richiamati degli Intent Receiver: il sistema avvierà l'applicazione quando uno di questi verrà innescato.
Come per ogni altro elemento di un'applicazione Android, anche gli Intent vanno registrati nel Manifest.

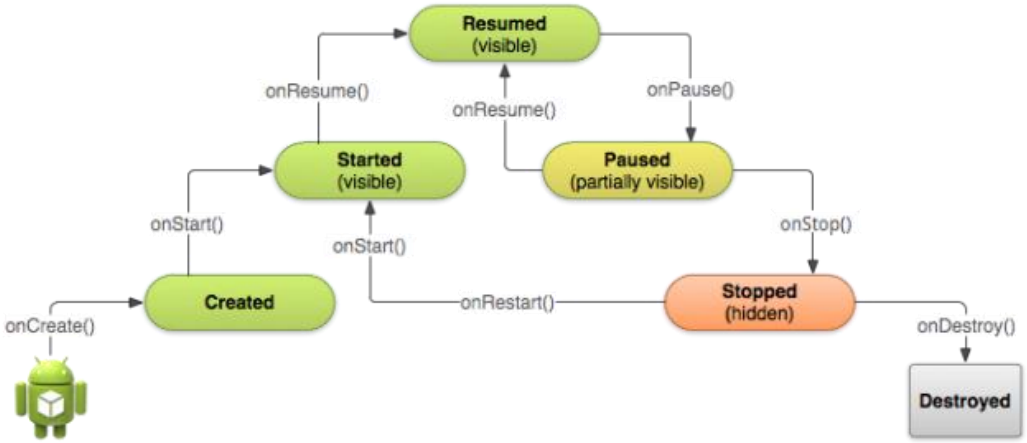
SERVICE

Un Service (servizio) è del codice utilizzato per gestire l'applicazione quando si trova in background, ovvero quando è chiusa o lo schermo è bloccato. Non presenta interfaccia utente.
Verrà analizzato più approfonditamente in seguito.

CONTENT PROVIDER

Le applicazioni possono immagazzinare i loro dati in diversi modi. Tra questi, troviamo la classe Content Provider. Questa implementa un set standard di metodi che permettono ad altre applicazioni di immagazzinare e recuperare i dati gestiti da uno stesso Content Provider.
E' utile quando si desidera che i dati dell'applicazione vengano condivisi con altre applicazioni.

CICLO DI VITA DI UN'APPLICAZIONE



Quando un'Activity entra in esecuzione, vengono sempre invocati tre metodi:

- *onCreate()*: l'Activity viene creata. Questa funzione viene chiamata ogni qual volta si è usciti dall'applicazione e la si riapre. In essa programmatore deve impostare la configurazione di base e definire quale sarà il layout dell'interfaccia;
- *onStart()*: l'Activity diventa visibile. È il momento in cui si possono attivare funzionalità e servizi che devono offrire informazioni all'utente;
- *onResume()*: l'Activity viene ripresa: l'applicazione non è stata chiusa, ma l'Activity non era aperta.

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

- *onPause()*: notifica la cessata interazione dell'utente con l'activity;
- *onStop()*: segna la fine della visibilità dell'activity;
- *onDestroy()*: segna la distruzione dell'activity.

Questi metodi sono concepiti a coppie, un metodo di avvio con un metodo di arresto: onCreate–onDestroy, onStart–onStop, onResume–onPause. Solitamente il lavoro fatto nel metodo di avvio – in termini di funzionalità attivate e risorse allocate – verrà annullato nel corrispondente metodo di arresto.

STRUTTURA DEL PROGETTO

La nostra applicazione è così strutturata:

- *AndroidManifest.xml*

Classi per la gestione del database:

- *DBHelper*
- *entityParola*

Activity, e i rispettivi layout, contenuti nella cartella res:

- *actAggiungiParole*
  res > layout > *activity_act_aggiungi_parole.xml*
- *actModificaParola*
  res > layout > *activity_act_modifica_parole.xml*
- *actIndex*
  res > layout > *activity_act_index.xml*

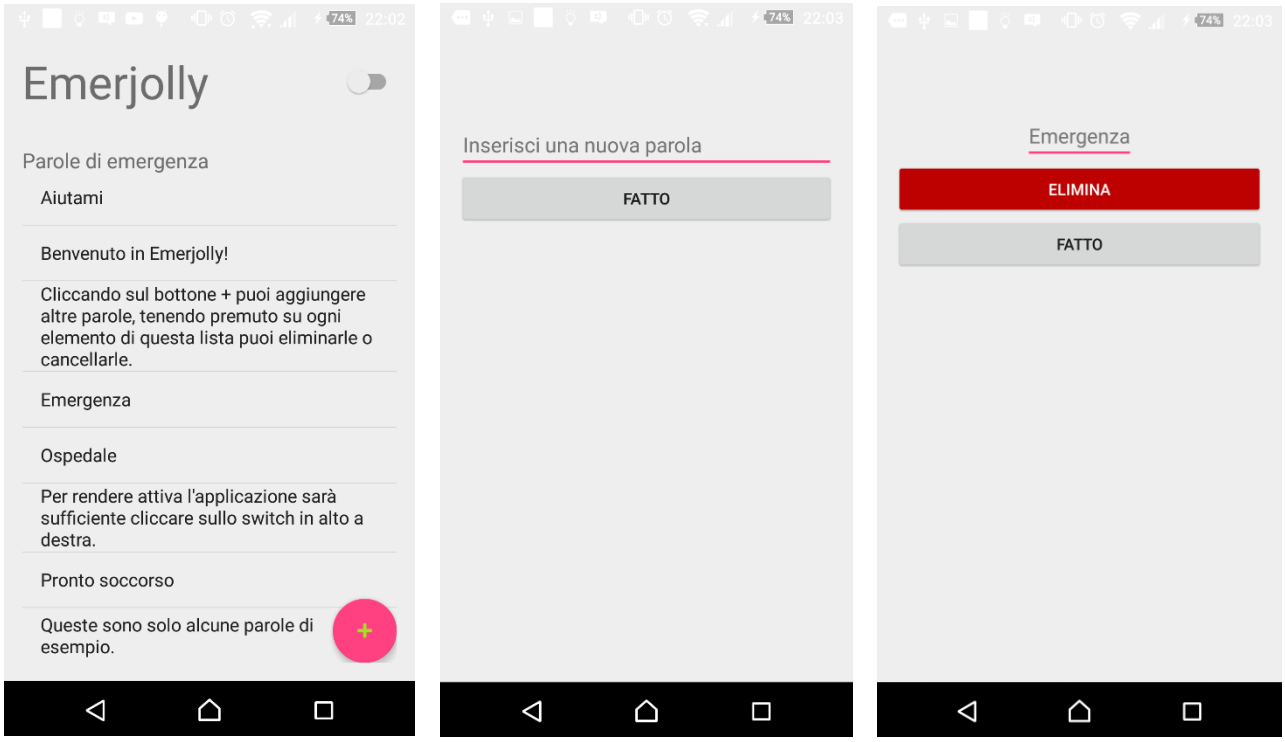
Classi MyService e SmsBroadcastReceiver, che permettono il background e la lettura degli SMS:

- *MyService*
- *SmsBroadcastReceiver*

Inoltre in *res > values > strings* abbiamo inserito tutte le stringhe necessarie ai vari layout, così in caso di modifiche o porting dell'applicazione in altre lingue sarebbe necessario modificare un solo file anziché tutto il codice:

```
<resources>
  <string name="app_name">Emerjolly</string>

  <string name="txt_parole_emergenza">Parole di emergenza</string>
  <string name="btn_aggiungi">+</string>

  <string name="txt_aggiungi_parole">Aggiungi parole</string>
  <string name="txt_gestione_parole">Modifica parole</string>
  <string name="editTxt_parola_scelta">Parola scelta</string>
  <string name="btn_elimina">Elimina</string>
  <string name="btn_fatto">Fatto</string>

  <string name="btn_stop">STOP</string>
</resources>
```

Esteticamente si presenta nel seguente modo:



ANDROID MANIFEST

Come già detto, il Manifest è un file XML in cui sono dichiarati tutti i permessi richiesti dall'applicazione, i suoi componenti, le relazioni tra di essi, le loro funzionalità e i requisiti.
Emerjolly richiede i permessi per leggere e ricevere gli sms, e per cambiare le impostazioni di sistema così da poter modificare l'audio.
Inoltre stabiliamo Index come parentActivity per AggiungiParole e ModificaParole: senza di questo, cliccare il tasto indietro chiuderebbe completamente l'applicazione.

MEMORIZZARE I DATI

Esistono diversi modi per memorizzare dati in Android. Nessuna soluzione è la migliore, dipende infatti dal contesto e dalla situazione in cui ci troviamo (per esempio possamo avere la necessità di tenere i dati privati e visibili solo alla nostra applicazione, oppure di renderli accessibili da altre applicazioni) e da quanto spazio i dati occupano.

- **Memorizzazione in locale**
  I dati sono memorizzati manualmente in dei file, salvati nella memoria del telefono (che può essere interna, esterna integrata o esterna)

- **Memorizzazione sul proprio server**
  Connettendoci al nostro server (di casa, dell'ufficio, ecc.), carichiamo i dati in un database.

- **Shared Preferences**
  I dati sono in numero limitato, primitivi, di piccole dimensioni. Vengono memorizzati in coppie chiave-valore.

- **Database**
  Dobbiamo gestire più entità i cui dati sono in grande quantità e complessi. Necessitano inoltre di essere in relazione tra loro, e di poter essere riordinati, selezionati, raggruppati con facilità. Per esempio, una rubrica telefonica di dimensioni realistiche utilizza un database.
  SQLlite è già integrato nel pacchetto di installazione di Android. Un database può essere visto solo dall'applicazione da cui è stato creato. Esistono comunque altri tipi di database (per esempio Realm.io, ORMlite), ma solitamente sono più complessi da utilizzare. Conviene utilizzarli per gestire grandi quantità di dati.

La nostra scelta poteva ricadere o sulle Shared Preferences, o sul database SQLite. Abbiamo immediatamente scartato la memorizzazione in locale perché avrebbe significato gestire creazione, lettura, scrittura e cancellazione di file XML, mentre le Shared Preferences e il database gestiscono autonomamente i file contenenti i dati. La memorizzazione su un nostro server non era possibile perché non lo possediamo.
Abbiamo quindi analizzato vantaggi e svantaggi delle soluzioni rimanenti:

	Shared Preferences	Database SQLite
Cosa sono	Le Shared Preferences sono coppie chiave-valore (un po' come un dizionario). Per accedere al dato è necessario conoscerne la chiave. Sono pensate per piccole quantità di dati, solitamente di tipo primitivo (boolean, String, int, long, float). Solitamente vengono utilizzate per salvare le preferenze dell'utente (per esempio la lingua), ma nulla vieta di salvare qualsiasi tipo di dati.	SQLite è una libreria che implementa un DBMS SQL molto veloce e compatto, ideale da utilizzare all'interno di dispositivi mobili. Il database è pensato per grandi quantità di dati. Non abbiamo optato per altri database, perché troppo complessi da integrare, specialmente considerando la piccola quantità di dati di cui abbiamo bisogno.
Punti in comune	❖ Offrono un'ampia gamma di metodi per lavorare sui dati. ❖ Sono integrati in Android Studio, e completamente supportati da esso. Non serve quindi scaricare librerie esterne. ❖ Entrambi scrivono su file di testo salvati nella memoria del telefono. ❖ Sono visibili solo dall'applicazione in cui vengono creati. Permettono di condividere i dati tra tutte le activity e le classi, evitando quindi avere dati ridondanti passati da un'Activity all'altra tramite gli Intent, e che appesantiscono la nostra applicazione. ❖ Esistono molte librerie aggiuntive per rendere più semplice l'utilizzo di entrambe le soluzioni. ❖ Non vengono cancellati se l'applicazione viene chiusa manualmente. L'unico modo di cancellarli è, dal menù, andare in Impostazioni > Applicazioni > (App) > "Cancella dati".	
Vantaggi	❖ Veloci per piccole quantità di dati, e più adatte a mappare dati semplici (Stringhe, interi, …) ❖ Dall'API 23 (Marshmellow) in poi, le SharedPreferences sono automaticamente salvate sul Google Drive dell'utente, e vengono automaticamente ripristinate quando si reinstalla l'applicazione. ❖ Semplici da comprendere e da utilizzare	❖ Veloce per grandi quantità di dati. ❖ Permette di effettuare operazioni sui dati (select, join, order by, …) e di gestirli in maniera molto più flessibile. ❖ La lettura e la scrittura dei dati è più robusta (anche se più lenta).
Svantaggi	❖ Lente per grandi quantità di dati. ❖ Sono coppie chiave-valore, e questo diventa problematico per leggere grandi quantità di dati, perché bisogna definire una chiave per ogni valore. E' difficile ricordarsi la giusta chiave per ogni singolo dato, a meno che non si abbiano delle regole precise per nominarle. E' comunque possibile, per esempio, convertire un dato in stringa, e lavorare poi su questa (cosa che comunque non andrà a modificare le Shared Preferences).	❖ Lento per piccole quantità di dati ❖ La scrittura del codice è più prolissa. ❖ E' necessario avere una conoscenza almeno basilare di SQL. Nonostante possa essere svantaggioso per chi ci si approccia per la prima volta, è in realtà un vantaggio per il futuro, poiché SQL viene utilizzato in tantissimi contesti.

Nonostante la nostra iniziale indecisione, abbiamo optato per il **database SQLite**. Per la sola ed unica memorizzazione delle parole le Shared Preferences sarebbero state più che sufficienti, ma questo avrebbe impedito eventuali sviluppi futuri: infatti la gestione di molti dati diversi collegati tra loro sarebbe stata estremamente difficile.
E' anche vero che un database composto da una sola tabella "Parola" contenente gli attributi "ID" e "parola" è forse esagerato, ed inoltre è più lento delle Shared Preferences. Abbiamo considerato i seguenti vantaggi più importanti della velocità: se in futuro volessimo aggiungere, per esempio, degli utenti (ognuno dei quali ha scelto delle parole diverse), il database ci permetterebbe di farlo con semplicità. Inoltre era necessario poter riordinare le parole per mostrarle in ordine alfabetico.

Abbiamo comunque utilizzato le **Shared Preferences** per memorizzare lo stato dello *switch* e il primo accesso al database.

DATABASE SQLITE

SQLite è una libreria software scritta in linguaggio C che implementa un DBMS SQL. E' leggera e compatta, più veloce di MySQL, e supporta database fino a 2TB. Il codice è distribuito gratuitamente. E' anche vero che non permette di gestire i permessi di accesso, e non supporta alcuni costrutti SQL come RIGHT JOIN, FULL OUTER JOIN, né le istruzioni per modificare, cancellare e rinominare le colonne di una tabella.

Un database SQLite è un file, che quindi possiamo facilmente spostare e copiare senza conseguenze. In Android viene memorizzato nella directory */data/data/(nome applicazione che lo utilizza)/databases*.

La gestione viene semplificata da Android e dalla classe DBHelper, ma è comunque necessario conoscere i comandi SQL principali.

LA CLASSE "PAROLA" E LA CLASSE "DBHELPER"

La classe *Parola* verrà utilizzata per riempire la tabella *parola* nel database.

La tabella *parola* si presenta nel seguente modo:

ID	nome
1	Ospedale
2	Emergenza
3	Pronto soccorso

La classe *Parola* estende l'interfaccia BaseColumns, ed ha come attributi:

- Le stringhe PAROLA_NOME e PAROLA_ID, che contengono il nome degli attributi presenti nella tabella Parola.
- La stringa "nome" e l'"ID", che è di tipo *long* poiché è così che viene memorizzato nel database. Contengono le variabili vere e proprie.

La classe *DBHelper*, che estende la classe SQLiteOpenHelper, serve per la gestione del database. In essa abbiamo scritto i metodi per aggiungere, modificare, eliminare ed in generale gestire le parole, oltre ai metodi per creare il database.
Per la gestione delle parole abbiamo a disposizione vari metodi, ognuno dei quali effettua una *query* al database. In Android bisogna utilizzare la funzione *execSQL()* se la query non restituisce nulla, come quella per creare nuove tabelle, mentre *rawQuery()* va utilizzata in tutti gli altri casi.

I metodi a nostra disposizione sono:

- *getAllParola()*
  Ritorna un vettore di stringhe contenente le parole del database.
- *parolaInesistente()*
  Data una parola, controlla se è presente nel database.
  Ritorna *true* se non vi è presente, altrimenti *false.*
- *aggiungiParola()*
- *eliminaParola()*
- *modificaParola()*

```
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

import java.util.ArrayList;

public class DBHelper extends SQLiteOpenHelper {
  private static final int DATABASE_VERSION = 1;
  private static final String DATABASE_NAME = "emerjolly";
  static final String TABLE_PAROLA = "parola";

  public DBHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
      //Creo le tabelle parola, parola_speciale, contatto, contatto_parola
      String query = "CREATE TABLE " + TABLE_PAROLA + "("
              + entityParola.PAROLA_ID + " integer primary key autoincrement, "
              + entityParola.PAROLA_NOME + " text not null);";

      db.execSQL(query);
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
      // Cancella la vecchia tabella
      db.execSQL("DROP TABLE IF EXISTS " + TABLE_PAROLA);
      // La ricrea
      onCreate(db);
  }

  //Ritorna un vettore di stringhe contenente le parole del database.
  public ArrayList<String> getAllParola() {
      ArrayList<entityParola> listaParole = new ArrayList<entityParola>();
      ArrayList<String> dbParole = new ArrayList<>();

      String query = "SELECT * FROM " + TABLE_PAROLA
              + " ORDER BY LOWER (nome)";

      SQLiteDatabase db = this.getReadableDatabase();
      Cursor cursor = db.rawQuery(query, null);
      if (cursor.moveToFirst()) {
          do { //Se la tabella ritornata dalla query non è vuota, la scorro riga per riga
              // e creo una nuova entityParola per ogni parola presente nel database.
              entityParola p = new entityParola();
              p.setID(Integer.parseInt(cursor.getString(0)));
              p.setParola(cursor.getString(1));
              listaParole.add(p);
          } while (cursor.moveToNext());
      }
      db.close();
```

```
        //Riempio il vettore dbParole con le stringhe contenute in ogni oggetto
entityParola.
        int i = 0;
        for (entityParola p : listaParole) {
            dbParole.add(i, p.getParola());
            i++;
        }
        return dbParole;
    }

    //Ritorna TRUE se la parola non è presente nel database.
    //Ritorna FALSE se, invece, è presente.
    public boolean parolaInesistente(entityParola p) {
        String query = "SELECT * FROM " + TABLE_PAROLA
                + " WHERE LOWER (nome) = LOWER ('" + p.getParola() + "')";
        //La query restituisce una tabella contenente ogni riga di TABLE_PAROLA
        // che contiene la parola P da noi passata.
        SQLiteDatabase db = this.getWritableDatabase();
        Cursor cursor = db.rawQuery(query, null);
        if (cursor.moveToFirst()) {
            return false;
        }
        db.close();
        return true;
    }

    //Aggiunge una nuova parola
    public void aggiungiParola(entityParola p) {
        SQLiteDatabase db = this.getWritableDatabase();

        ContentValues values = new ContentValues();
        values.put(entityParola.PAROLA_NOME, p.getParola());
        long id = db.insert(TABLE_PAROLA, null, values); //il metodo insert() restituisce
un long corrispondente all'ID dell'oggetto nel database
        p.setID(id);
        db.close();
    }

    //Elimina una parola
    public void eliminaParola(entityParola p) {
        SQLiteDatabase db = this.getWritableDatabase();
        db.delete(TABLE_PAROLA, entityParola.PAROLA_NOME + "=?", new
String[]{p.getParola()});
        db.close();
    }

    //Modifico una parola, eliminando quella vecchia e inserendo quella nuova
    public void modificaParola(entityParola vecchia, entityParola nuova) {
        SQLiteDatabase db = this.getWritableDatabase();

        this.eliminaParola(vecchia);
        this.aggiungiParola(nuova);
    }
}
```

SHARED PREFERENCES

La classe *SharedPreferences* ci fornisce tutti gli strumenti di cui abbiamo bisogno. I dati collocati nelle preferenze vengono salvati in un file XML nella cartella *shared_prefs*, che risiede nella memoria interna del telefono.
L'**inizializzazione** di un oggetto può avvenire in due modi:

- *getPreferences(int mode)*
  Da utilizzare se utilizziamo un unico file per tutte le preferenze. Necessita solo della modalità di accesso.
- *getSharedPreferences(String name, int mode)*;
  Metodo da utilizzare se necessitiamo di più file per le preferenze. Specifichiamo il nome di ogni file nel primo parametro. Il secondo parametro, invece, indica la modalità di accesso (per rendere le SharedPreferences private o accessibili a tutti).

Entrambi i metodi restituiscono un'istanza della classe SharedPreferences.

```
//Per impostare la modalità d'accesso, è possibile utilizzare sia 0 -
privata- o 1 -pubblica-, oppure ACCESS_MODE_PRIVATE /
ACCESS_MODE_PUBLIC. Queste ultime sono variabili di Android contenenti i
precedenti interi.
SharedPreferences myPrefs = getSharedPreferences("LeMiePreferenze", 0);
```

Esistono moltissimi metodi *getter* per la **lettura** dei dati: *getString()*, *getInt()*, *getBoolean()*, e così via. Ogni metodo richiedere due argomenti: la chiave utilizzata per memorizzare il dato, ed un valore da restituire in caso di dato non trovato.

```
String nome = myPrefs.getString("Nome", "Nome non trovato");
```

La **scrittura**, invece, richiede la creazione di un oggetto della classe *SharedPreferences.Editor*. Questo andrà a modificare i nostri dati utilizzando metodi tipo *setter* come *putBoolean()* o *putString()*, come avveniva per la lettura.
Per salvare le modifiche possiamo utilizzare *commit()* oppure *apply()*: la differenza è che il primo salva immediatamente i dati, il secondo gestisce il salvataggio in background.

```
SharedPreferences.Editor myEditor = myPrefs.edit();

myEditor.putString("Nome", "Paolina");

myEditor.commit();
```

Nella nostra applicazione, creiamo le SharedPreferences nel metodo *onCreate()*. Le utilizzeremo per memorizzare il primo accesso al database, e lo stato dello switch.
Desideriamo che al primo accesso dell'utente, egli possa vedere alcune stringhe di benvenuto, oltre a delle parole di default come "Ospedale", "Emergenza", … E' importante che questo accada solo ed unicamente al primo accesso: non quando il database è vuoto (perché l'utente potrebbe aver eliminato tutte le parole per aggiungerne altre), nè in nessun altro caso.
La soluzione è utilizzare le SharedPreferences nel seguente modo: controlliamo se al loro interno è presente la stringa "primoAccesso". Ovviamente al primo avvio dell'applicazione le preferenze sono vuote, quindi riempiamo il database con alcune parole di default, e aggiungiamo nelle preferenze la stringa "primoAccesso".
Se la stringa è presente (quindi dal secondo accesso in poi), invece, utilizziamo i metodi creati nella classe DBHelper per riempire il vettore **`parole`**.

```
// Se le SP non contengono "primoAccesso", riempio il db con parole di default,
// e poi inserisco primoAccesso nelle SP.
// Se le SP contengono "primoAccesso", svuoto il vettore di parole.
sharedPref = PreferenceManager.getDefaultSharedPreferences(this);
SharedPreferences.Editor editor = sharedPref.edit();
if (sharedPref.contains("primoAccesso")) {
  parole.clear();
} else {
  db.aggiungiParola(new entityParola("Benvenuto in Emerjolly!"));
  db.aggiungiParola(new entityParola("Queste sono solo alcune parole di esempio."));
  db.aggiungiParola(new entityParola("Cliccando sul bottone + puoi aggiungere altre
parole, " +
          "tenendo premuto su ogni elemento di questa lista puoi eliminarle o
cancellarle."));
  db.aggiungiParola(new entityParola("Per rendere attiva l'applicazione sarà sufficiente
cliccare sullo switch in alto a destra."));
  db.aggiungiParola(new entityParola("Ospedale"));
  db.aggiungiParola(new entityParola("Emergenza"));
  db.aggiungiParola(new entityParola("Pronto soccorso"));
  db.aggiungiParola(new entityParola("Aiutami"));
  editor.putBoolean("primoAccesso", true).apply();
}
//In entrambi i casi, riempio il vettore di parole con il contenuto del db.
parole = db.getAllParola();
```

Per quanto riguarda invece lo stato dello *switch*, alla creazione dell'Activity assegno alla variabile booleana *state* il valore salvato nelle preferenze con la chiave "state".
Nel metodo *getBoolean()* il primo parametro indica la chiave, il secondo indica il valore da ritornare in caso non esista nulla in corrispondenza di quella chiave: in questo modo *state* assume il valore false anche se non abbiamo mai toccato lo switch (e, di conseguenza, non c'è mai stato nessun cambiamento di stato da memorizzare).
Successivamente utilizziamo il metodo *setChecked()* per forzare l'aspetto grafico del bottone: se *state* corrisponde a true, apparirà acceso, altrimenti spento.
Solo quando tocchiamo lo switch inseriamo un valore sotto alla chiave "state": *true* se è acceso, *false* se è spento.

```
//Assegno alla variabile "state" lo stato dello Switch salvato nelle Shared Preferences
boolean state = sharedPref.getBoolean("state", false);
switchOnOff.setChecked(state);
switchOnOff.setOnClickListener(new Switch.OnClickListener() {
  @Override
  public void onClick(View v) {
      if (switchOnOff.isChecked()) {
          sharedPref.edit().putBoolean("state", true).apply();
          startService(new Intent(actIndex.this, MyService.class));
      } else {
          sharedPref.edit().putBoolean("state", false).apply();
          stopService(new Intent(actIndex.this, MyService.class));
      }
  }
});
```

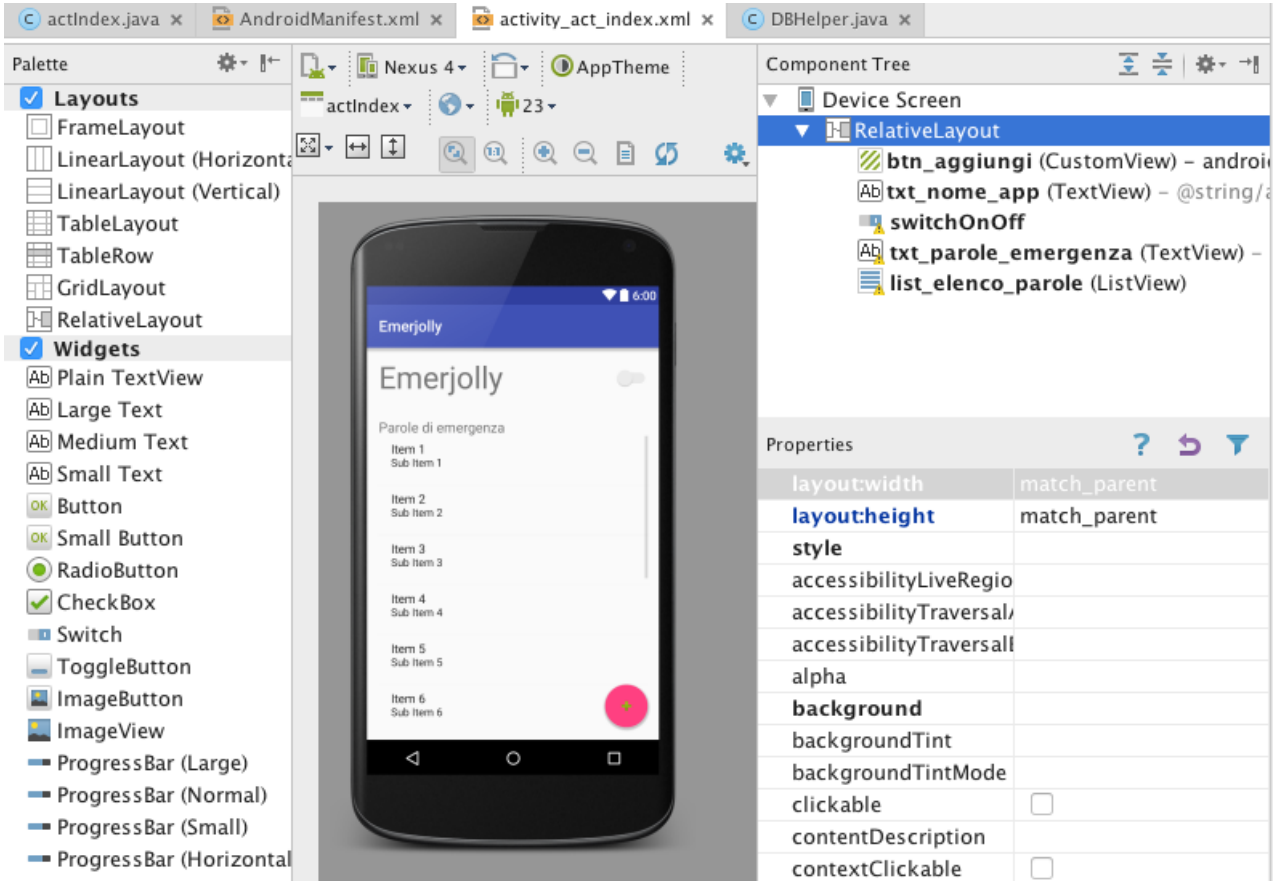
Il metodo *onResume()* aggiorna l'Activity con le ultime modifiche. Viene chiamato automaticamente ogni qual volta venga riaperta l'Activity senza essere usciti dall'applicazione, quindi per esempio tornando indietro da AggiungiParola o ModificaParola.
Se non esistesse questo metodo, nonostante il database e le preferenze si aggiornino immediatamente, la grafica dell'applicazione non mostrerebbe le nuove modifiche. Per esempio, una nuova parola appena aggiunta esisterebbe nel database, ma non sarebbe mostrata nella lista, e l'unico modo per vederla sarebbe chiudere a mano l'applicazione e poi riaprirla.

```
@Override
//Ogni volta che l'Activity viene "ripresa" chiama questa funzione.
public void onResume() {
  //Riempio il vettore di parole così da poter mostrare eventuali aggiornamenti nella
lista
  DBHelper db = new DBHelper(this);
  parole.clear();
  parole = db.getAllParola();

  //Imposto l'aspetto dello switch
  sharedPref = PreferenceManager.getDefaultSharedPreferences(this);
  boolean state= sharedPref.getBoolean("state", false);
  switchOnOff.setChecked(state);

  btn_stop.setVisibility(View.GONE);

  //Assegno il vettore di parole alla lista
  ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
          android.R.layout.simple_list_item_1, android.R.id.text1, parole);
  list_elenco_parole.setAdapter(adapter);
  super.onResume();
}
```

LE ACTIVITY

INDEX

Questa è la pagina iniziale. Contiene uno switch che, se cliccato, mette l'applicazione in background, una lista di parole modificabili ed eliminabili, un bottone per aggiungere nuove parole, ed un bottone per fermare la suoneria. Quest'ultimo normalmente è invisibile, si può vedere solo quando il telefono sta suonando, e nel momento in cui lo si clicca sparisce di nuovo.
Essendo questa la classe principale del progetto, in essa sono presenti anche le istruzioni per la gestione di database e Shared Preferences viste precedentemente, oltre alla funzione per il controllo degli SMS e il cambio della suoneria.
Il suo layout è definito nel file *activity_act_index.xml*.
Ogni elemento del layout può avere diversi attributi: il più importante è l'ID, che ci permetterà di identificare un determinato oggetto grafico anche nel codice.



Il codice è invece definito nella classe *actIndex*. Questa classe possiede metodi per gestire gli eventi scatenati dalle seguenti azioni:

- **Click sullo switch**
  Salvataggio del nuovo stato dello switch nelle Shared Preferences, ed avvio di un servizio in background (che verrà spiegato nei prossimi capitoli).
- **Click sul bottone per aggiungere nuove parole**
  Viene creato un Intent che porta all'Activity per aggiungere le parole.
- **Click lungo (cioè tenere premuto) su un qualsiasi elemento della lista**
  Viene creato un Intent che porta all'Activity per modificare ed eliminare le parole.

- **Click sul bottone di stop.**
  Questa funzione verrà contestualizzata meglio in seguito, poiché è all'interno della funzione per controllare i messaggi e far suonare il cellulare, argomenti trattati nei prossimi capitoli.

La prassi è sempre la stessa, per ogni elemento grafico e per ogni interazione possibile con esso (click, swipe, …): per prima cosa bisogna dichiarare una variabile di tipo Button, ListView, ecc... all'interno della classe, ma fuori dall'*onCreate()*. In quest'ultimo metodo bisogna assegnare la variabile ad un oggetto presente nel layout, tramite l'**ID**.
Successivamente, gli assegnamo un Listener, ovvero un oggetto che "ascolta" tutto ciò che l'utente fa, e che reagisce se egli esegue una determinata azione. Per esempio, l'*onClickListener* reagisce solo se l'oggetto viene cliccato.
Infine, utilizziamo funzioni come *onClick()*, *onLongClick()*, … per dire all'applicazione cosa fare in caso di interazione da parte dell'utente.

```
  btn_stop.setVisibility(View.GONE);

  final Intent intent1 = new Intent(this, actAggiungiParole.class);
  btn_aggiungi.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View view) {
          startActivity(intent1);
      }
  });


  // Definisco un nuovo Adapter per riempire la lista.
  // I parametri sono: context, layout per le righe,
  // ID della TextView nella quale verranno inserite le parole, e il vettore che le
contiene.
  ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
          android.R.layout.simple_list_item_1, android.R.id.text1, parole);
  list_elenco_parole.setAdapter(adapter);

  final Intent intent2 = new Intent(this, actModificaParole.class);
  list_elenco_parole.setOnItemLongClickListener(new AdapterView.OnItemLongClickListener()
{
      @Override
      public boolean onItemLongClick(AdapterView<?> parent, View view, int position, long
id) {
          String parola_scelta = (String)
(list_elenco_parole.getItemAtPosition(position));
          intent2.putExtra("parola_scelta", parola_scelta); //parola_scelta contiene la
parola selezionata dall'utente
          startActivity(intent2);
          return true;
      }
  });
}
```

AGGIUNGI PAROLE

Questa Activity, chiamata dall'onClick sul bottone in basso a destra in Index, permette di aggiungere delle nuove parole. Possiede un EditText (*id: editTxt_nuova*_parola) in cui poter scrivere, ed un bottone con scritto "Fatto" (*id: btn_fatto*).
Ogni volta che il bottone viene cliccato, il contenuto del campo di testo viene controllato: se è vuoto, compare un messaggio di errore. Se, invece, vi è stata inserita una parola, controlliamo se è già presente nel database. In caso affermativo avvisiamo l'utente, altrimenti la aggiungiamo.

```
        editTxt_nuova_parola = (EditText) findViewById(R.id.editTxt_nuova_parola);
        btn_fatto = (Button) findViewById(R.id.btn_fatto);
        btn_fatto.setOnClickListener(new Button.OnClickListener() {
            @Override
            public void onClick(View v) {
                String s = editTxt_nuova_parola.getText().toString().trim(); //Salvo la
nuova parola in 's'
                boolean flag = true;
                if (s.isEmpty() || s.equals("")) { //Controllo se la nuova parola è
vuota
                    AlertDialog.Builder alert = new
AlertDialog.Builder(actAggiungiParole.this);
                    alert.setTitle("Attenzione");
                    alert.setMessage("Controlla la parola inserita");
                    alert.setPositiveButton("Ok", null);
                    alert.show();
                    flag = false; //Se il flag viene cambiato in false significa che la
nuova parola è vuota, quindi non entrerà nel prossimo if e non la aggiungerà al database
                }
                if (flag == true) {
                    entityParola p = new entityParola(s); //Creo una parola contenente
la nuova parola

                    if (db.parolaInesistente(p)) { //Se la parola non esiste già, la
aggiungo
                        db.aggiungiParola(p);
                        finish();
                    } else { //Altrimenti, avviso l'utente
                        AlertDialog.Builder alert = new
AlertDialog.Builder(actAggiungiParole.this);
                        alert.setTitle("Attenzione");
                        alert.setMessage("Parola già esistente");
                        alert.setPositiveButton("Ok", null);
                        alert.show();
                    }
                }
            }
        });
```

MODIFICA PAROLE

Questa Activity, chiamata dall'onLongClick su qualsiasi elemento della lista, permette di modificare o eliminare le parole già esistenti. Possiede un EditText (*id: editTxt_parola_scelta*) in cui poter scrivere, e due bottoni, uno con scritto "Fatto" (*id: btn_fatto*) ed uno per l'eliminazione (*id: btn_elimina*).
Quando l'Activity viene creata in *actIndex*, gli viene aggiunto un extra, contenente la parola scelta. Questa verrà memorizzata nella stringa **`parola_scelta`**.

Abbiamo due funzioni: una per l'eliminazione della parola scelta, ed una per la sua modifica.
Mentre la prima utilizza semplicemente il metodo del database *eliminaParola()*, la seconda invece prima memorizza il contenuto dell'editTxt, cioè la nuova parola scelta dall'utente, in una stringa, poi controlla se è vuota ed infine se è già presente nel database.

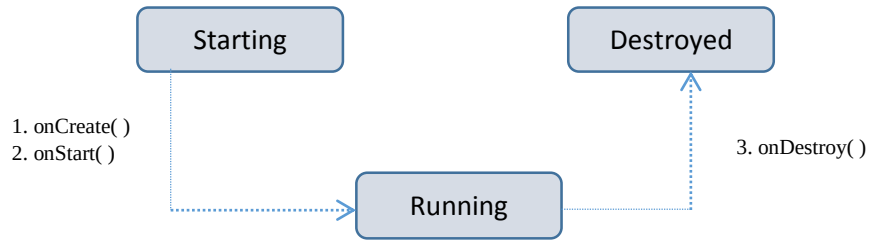
```
    if (getIntent().getExtras() != null)
```

```
        parola_scelta = getIntent().getStringExtra("parola_scelta").trim(); //Memorizzo
la parola scelta dall'utente
    editTxt_parola_scelta.setHint(parola_scelta); //e la mostro nell'edit text

    btn_elimina.setOnClickListener(new Button.OnClickListener() {
        @Override
        public void onClick(View v) {
            db.eliminaParola(new entityParola(parola_scelta));
            finish();
        }
    });

    btn_fatto.setOnClickListener(new Button.OnClickListener() {
        @Override
        public void onClick(View v) {
            String s = editTxt_parola_scelta.getText().toString().trim(); //Salvo la
nuova parola inserita dall'utente
            boolean flag = true;

            if (s.isEmpty() || s.equals("")) {
                AlertDialog.Builder alert = new
AlertDialog.Builder(actModificaParole.this);
                alert.setTitle("Attenzione");
                alert.setMessage("Controlla la parola inserita");
                alert.setPositiveButton("Ok", null);
                alert.show();
                flag = false;
            }

            if (flag == true) {
                //Creo due oggetti parola, uno per la vecchia parole ed uno per la
nuova
                entityParola vecchia = new entityParola(parola_scelta);
                entityParola nuova = new entityParola(s);

                if (db.parolaInesistente(nuova)) {
                    db.modificaParola(vecchia, nuova);
                    finish();
                } else {
                    AlertDialog.Builder alert = new
AlertDialog.Builder(actModificaParole.this);
                    alert.setTitle("Attenzione");
                    alert.setMessage("Parola già esistente");
                    alert.setPositiveButton("Ok", null);
                    alert.show();
                }
            }
        }
    });
```

SERVICE

ESEGUIRE OPERAZIONI IN BACKGROUND

Un Service è un componente fondamentale di un'app Android: è utilizzato per eseguire un'operazione caratterizzata da una lunga durata di esecuzione e dalla mancanza di interazione diretta con l'utente. Esso, a differenza delle Activity, non utilizza lo schermo del dispositivo e viene quindi impiegato per gestire operazioni da effettuare in background.
Il ciclo di vita di un Service è molto più semplice rispetto a quello delle Activity. Un Service può essere avviato (start) o interrotto (stop) ed è controllato più dallo sviluppatore che dal sistema.



La classe Service generica ha alcuni metodi fondamentali, che è necessario ridefinire per realizzare uno specifico Service che svolga le funzioni da noi desiderate.

ONCREATE( )	Il sistema operativo invoca questo metodo quando il Service viene creato per la prima volta, prima di invocare onStartCommand() o onBind(); se il service è già in esecuzione questo metodo non viene invocato.
ONSTARTCOMMAND( )	Il sistema operativo invoca questo metodo quando un altro componente richiede l'esecuzione del Service invocando il metodo startService(). Il Service non viene interrotto finché un altro componente non invoca il metodo stopService(), il service invoca il metodo stopSelf(), oppure il sistema operativo lo distrugge per insufficienza di memoria.
ONBIND( )	Il sistema operativo invoca questo metodo quando un altro componente richiede di collegarsi al Service per effettuare una comunicazione client/server invocando il metodo bindService(). È obbligatorio implementare questo metodo, ma se non si è interessati a far comunicare il service con altri componenti è sufficiente eseguire *return null*. Successivamente all'esecuzione del metodo il Service viene distrutto dal sistema operativo.
ONDESTROY( )	Il sistema operativo invoca questo metodo quando il Service deve essere distrutto: bisogna ridefinire questo metodo per distruggere oggetti o thread creati dal servizio. Si tratta dell'ultimo metodo invocato prima della distruzione definitia di un Service.

Un Service può essere implementato nel thread principale dell'applicazione, o in un nuovo thread. La decisione è a discrezione del programmatore.
E' obbligatorio dichiarare ogni Service nel file Manifest dell'app: a questo scopo è sufficiente aggiungere un tag <service> all'interno dell'elemento <application>, ricordando che l'unico attributo richiesto è il nome della classe che implementa il Service.

```
<manifest ...>

  ...

  <application ... >

      <service
          android:name=".MyService"
          ...
          />
  ...

  </application>

</manifest>
```

SMS BroadcastReceiver

Come è stato detto precedentemente il BroadcastReceiver viene utilizzato quando vogliamo che del codice dell'applicazione si esegua solo in relazione ad un evento esterno, nel nostro caso la ricezione di un nuovo SMS.
Analizziamo il messaggio e lo inseriamo in una stringa di testo, che successivamente verrà confrontata con tutte le parole d'emergenza.

La nostra applicazione per riuscire ad intercettare un SMS ed analizzarlo deve avere dei permessi speciali che saranno mostrati all'utente al momento dell'installazione.
Questi vanno dichiarati nel file XML *AndroidManifest* e sono:

- Permesso per leggere gli SMS ed analizzarli

  ```
  <uses-permission android:name="android.permission.READ_SMS" />
  ```

- Permesso per intercettare l'SMS

  ```
  <uses-permission android:name="android.permission.RECEIVE_SMS" />
  ```

MODIFICA DELL'AUDIO

E' fondamentale che la nostra applicazione suoni sempre, anche se il telefono è in vibrazione o in silenzioso. L'unico modo per farlo è modificare le impostazioni di sistema riguardanti il suono, quindi aggiungiamo il permesso `WRITE_SETTINGS` al manifest.

Per la modifica dell'audio, utilizziamo le classi RingtoneManager, Ringtone e AudioManager.

RINGTONE MANAGER

La classe **RingtoneManager** ci permette di **accedere** alle suonerie, alle notifiche e ai vari suoni di sistema. Può inoltre restituire un oggetto di classe *Ringtone* per ogni suoneria. La classe *Ringtone* offre dei metodi per far partire una notifica o una suoneria.
Ogni istanza della classe *RingtoneManager* ha a disposizione un'infinità di metodi e di costanti, quelli da noi utilizzati sono:

TIPO	COSTANTE	AZIONE

int	`TYPE_NOTIFICATION (2)`
	`TYPE_ALARM (4)`
	`TYPE_RINGTONE (1)`
	`TYPE_ALL (7)`
		Questo intero indica il "tipo" dei file utilizzati per le notifiche. Serve al sistema operativo per sapere quali file rendere disponibili quando si sta cambiando il suono delle notifiche, delle sveglie, o della suoneria. Tra parentesi sono indicati i valori numerici a cui queste costanti corrispondono.

METODO	DATO DI RITORNO	AZIONE
`getDefaultUri(int)`	static URI	Ritorna l'**URI** della suoneria di default di un particolare "tipo" (quindi passando TYPE_ALARM otterremo l'URI della sveglia che adesso l'utente utilizza). Un **URI** (***U**niform **R**esource **I**dentifier*) è una stringa univoca che identifica qualsiasi risorsa, sia sul web sia sul dispositivo locale.
`getRingtone(Context c, Uri ringtoneUri)`	static Ringtone	Ritorna l'oggetto Ringtone corrispondente all'URI passato alla funzione.

RINGTONE

Come già detto, la classe *Ringtone* ci permette di **far partire** una suoneria.
A differenza della classe *RingtoneManager*, qui abbiamo a disposizione solo metodi, e nessuna costante.
Alcuni esempi sono:

```
Ringtone ringtone = RingtoneManager.getRingtone(getApplicationContext(), alert);
ringtone.setStreamType(AudioManager.STREAM_ALARM);
```

METODO	DATO DI RITORNO	AZIONE
`setStreamType(int)`	*void*	Imposta il tipo di suoneria. Bisogna passarvi una costante della classe AudioManager, come STREAM_ALARM per le suonerie disponibili per la sveglia, o STREAM_RINGTONE per quelle delle chiamate.
`getTitle()`	String	Ritorna il titolo del file in un formato adatto.

AUDIO MANAGER

Come già detto, la classe *AudioManager* permette **l'accesso al volume** e, di conseguenza, la sua **modifica**.

TIPO	COSTANTE	AZIONE
String	`STREAM_ALARM`	L'audio utilizzato per le sveglie.

int	`TYPE_NOTIFICATION (2)`
	`TYPE_ALARM (4)`
	`TYPE_RINGTONE (1)`
	`TYPE_ALL (7)`
		Questo intero indica il "tipo" dei file utilizzati per le notifiche. Serve al sistema operativo per sapere quali file rendere disponibili quando si sta cambiando il suono delle notifiche, delle sveglie, o della suoneria. Tra parentesi sono indicati i valori numerici a cui queste costanti corrispondono.

METODO	DATO DI RITORNO	AZIONE
`getStreamVolume (int)`	int	Ritorna il valore del volume di sistema. In silenzioso o in vibrazione il valore corrisponde a 0.
`getStreamMaxVolume`	int	Ritorna il valore massimo che può avere il volume.

```
public void updateList(final String smsMessage, String address) throws
InterruptedException {
  sms.add(0, smsMessage);
  String s=smsMessage.toLowerCase();

  for (int i = 0; i < parole.size(); i++) {
      if (s.contains(parole.get(i).trim().toLowerCase())) {
          //Ottengo l'URI della suoneria predefinita per la sveglia
          Uri alert = RingtoneManager.getDefaultUri(RingtoneManager.TYPE_ALARM);
          //Se non c'è, tento di ottenere l'URI della suoneria predefinita delle
notifiche, o delle chiamate
          if (alert == null) {
              alert = RingtoneManager.getDefaultUri(RingtoneManager.TYPE_NOTIFICATION);
              if (alert == null) {
                  alert = RingtoneManager.getDefaultUri(RingtoneManager.TYPE_RINGTONE);
              }
          }
          //Ottengo l'oggetto Ringtone corrispondente all'URI
          final Ringtone ringtone = RingtoneManager.getRingtone(getApplicationContext(),
alert);
```

```
            //Imposto il tipo di suoneria come 'ALARM' cioè la sveglia
            ringtone.setStreamType(AudioManager.STREAM_ALARM);
            //Ottengo un oggetto audioManager che mi permette di gestire le impostazioni
riguardanti l'audio del telefono
            AudioManager audioManager = (AudioManager)
getSystemService(Context.AUDIO_SERVICE);
            //Da esso leggo il volume attuale del telefono
            int volume = audioManager.getStreamVolume(AudioManager.STREAM_ALARM);
            if (volume == 0) {
                //Se il volume è == 0 (silenzioso o vibrazione) salvo in 'volume' il valore
massimo che può avere il volume
                volume = audioManager.getStreamMaxVolume(AudioManager.STREAM_ALARM);
            }
            //Imposto la suoneria, il volume del telefono, e rimuovo la vibrazione
            audioManager.setStreamVolume(AudioManager.STREAM_ALARM, volume,
AudioManager.FLAG_REMOVE_SOUND_AND_VIBRATE);

            if (ringtone != null) {
                ringtone.play;

                //Ogni elemento grafico viene reso non solo invisibile, ma lo spazio da esso
                occupato può essere occupato con altri oggetti.
                audioManager.setStreamVolume(0, 0, AudioManager.FLAG_VIBRATE);
                switchOnOff.setVisibility(View.GONE);
                list_elenco_parole.setVisibility(View.GONE);
                btn_aggiungi.setVisibility(View.GONE);

                //Al loro posto viene infatti mostrato il bottone per fermare la suoneria.
                btn_stop.setVisibility(View.VISIBLE);
                btn_stop.setOnClickListener(new View.OnClickListener() {
                    @Override
                    public void onClick(View view) {
                        ringtone.stop();
                        btn_stop.setVisibility(View.GONE);
                        switchOnOff.setVisibility(View.VISIBLE);
                        list_elenco_parole.setVisibility(View.VISIBLE);
                        btn_aggiungi.setVisibility(View.VISIBLE);
                    }
                });
            }
        }
    }
}
```

DISTRIBUIRE UN'APPLICAZIONE

PREPARAZIONE DEI FILE

Una volta sicuri che la nostra applicazione sia priva di bug, dobbiamo disabilitare l'opzione *debuggable* e apportare alcune modifiche:

- Nel file *Project Attributes* dobbiamo aggiungere o controllare che il codice della versione dell'applicazione sia corretto: questo viene utilizzato dalla piattaforma Android per gestire gli aggiornamenti.

  ```
  #VersionCode: 1
      #VersionName: 1.00
  ```

- Nel Manifest dobbiamo aggiungere l'elemento ***<uses-sdk>*** per specificare il range di versioni Android in cui l'applicazione può funzionare. E' necessario indicare la minima, la massima e la principale versione. Dobbiamo anche aggiungere il tag ***<permission>***, che indica i permessi richiesti dall'applicazione. Ovviamente questi devono essere limitati al minimo, non bisogna richiedere permessi non necessari.
  Infine aggiungiamo il tag ***<uses-feature>***, che specifica le caratteristiche hardware o software necessarie perché l'applicazione funzioni.

Siamo quindi pronti a creare l'APK della nostra applicazione.

COS'È UN APK?

« L'estensione *apk* indica un file Android Package, variante del formato .JAR, ed è utilizzato per la distribuzione e l'installazione di componenti in dotazione sulla piattaforma per dispositivi mobili Android. »
(fonte: Wikipedia)

Gli APK sono quindi dei file di installazione da cui è possibile estrarre l'applicazione vera e propria. Per crearli è sufficiente andare nella cartella dove abbiamo salvato il progetto, e poi selezionare Build > ouputs > apk, e selezionare il file con l'estensione .apk. E' importante aver cambiato la modalità da "Debug" a "Release".
E' buona norma apporvi una *firma digitale*, soluzione possibile sia utilizzando software di terze parti, oppure utilizzando la funzione di Android Studio Build > Genera APK firmato.
E' importante che la chiave di cifratura scada tra molto tempo, poiché alla sua scadenza gli utenti non saranno più in grado di aggiornare l'app.
La firma digitale viene controllata ogni volta che è necessario un aggiornamento: se il certificato della nuova versione è uguale a quello della vecchia versione, il sistema permette l'aggiornamento. In caso contrario, è necessario che lo sviluppatore cambi il nome dell'applicazione, che verrà installata come nuova.
Inoltre le applicazioni che utilizzano lo stesso certificato possono utilizzare lo stesso processo, e condividere dati e codice solo tra applicazioni con certificato uguale al proprio.

DISTRIBUZIONE AL PUBBLICO

La distribuzione può avvenire in tre modi:

- Manualmente, ad esempio, inviando l'apk tramite mail.
  Può essere utile se, per qualsiasi motivo, non desideriamo realmente che la nostra applicazione sia a disposizione di chiunque.
- Esporre l'applicazione su un server web.
  L'applicazione sarà disponibile a chiunque desideri scaricarla. Non viene controllata, quindi è possibile che malintenzionati utilizzino questo metodo per diffondere virus o malware. Inoltre la pubblicazione "in proprio" non garantisce una grande visibilità, dal momento che in Android il punto di riferimento per il download di applicazioni è il Google Play Store.
- Pubblicare l'applicazione sul Google Play Store.

PUBBLICAZIONE SU GOOGLE PLAY

Google Play è un vantaggio sia per lo sviluppatore, poiché è lo store più utilizzato, sia per l'utente finale, perché ogni applicazione deve superare rigidi controlli per essere approvata e pubblicata e questo garantisce una certa difesa dai malintenzionati.
Per prima cosa, bisogna registrarsi come sviluppatore su Google Play. A differenza degli sviluppatore iOS, che devono rinnovare il pagamento di $99 ogni anno, Google richiede solo un pagamento di $25 all'iscrizione.
Le regole previste da Google per la richiesta di pubblicazione dell'APK sono le seguenti:

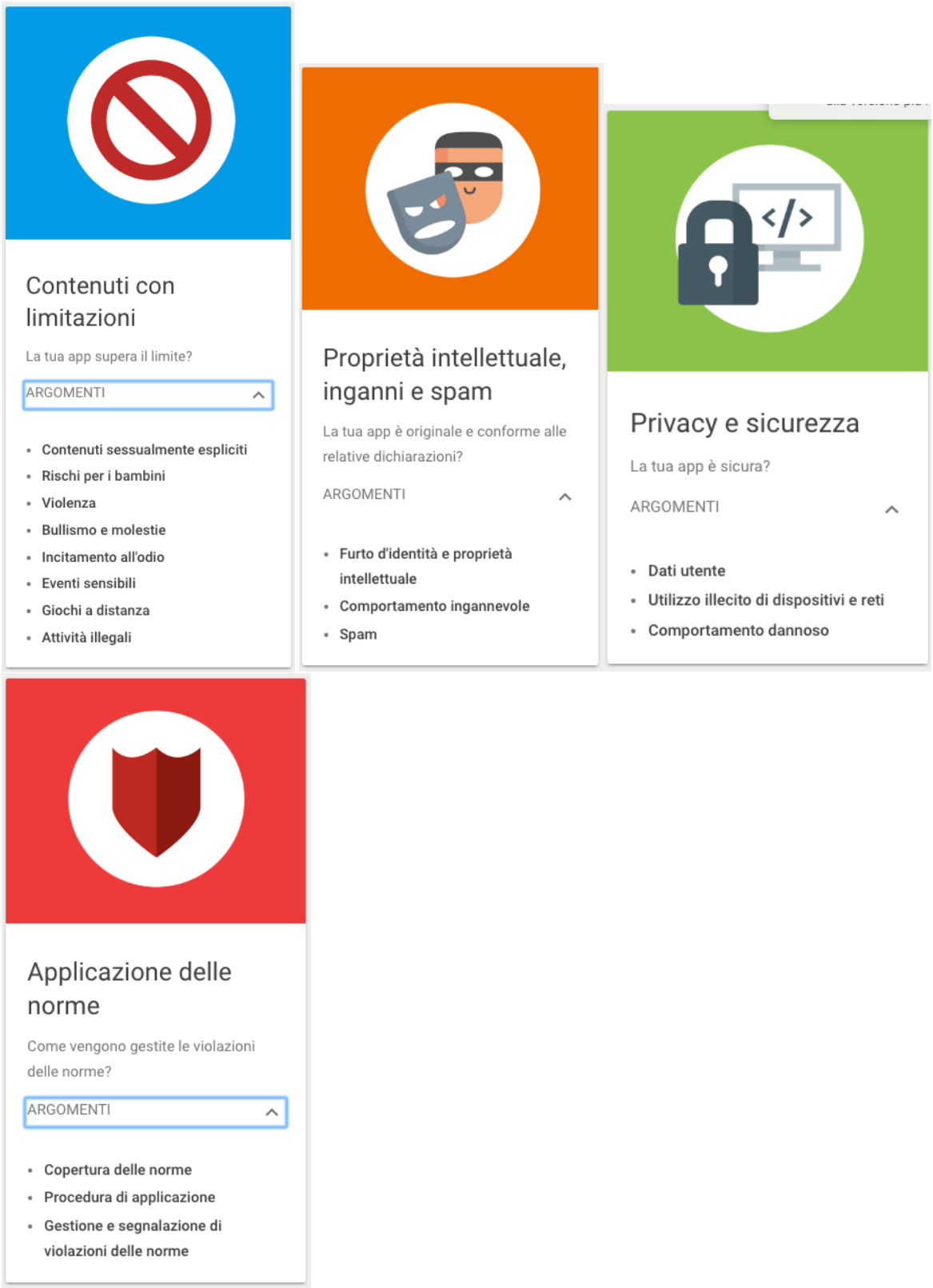
- I files non devono superare i 50 Mb
- Bisogna aggiungere almeno due screenshot (sui sei totali)
- L'icona dell'applicazione deve essere in alta definizione
- È obbligatorio inserire una descrizione.
- Bisogna selezionare una categoria, una classificazione dei contenuti, almeno un Paese di destinazione, e un URL delle norme sulla privacy.

Rispettare queste regole ci permette di candidare la nostra app per la pubblicazione, ma non ci garantisce che verrà effettivamente pubblicata. Infatti, prima subirà un rigido controllo, e solo se rispetterà tutte le regole di Google potremo vederla sullo Store entro le 12 ore seguenti. Alcune di queste regole sono:

- E' proibito pubblicare applicazioni con icona e nome simile a quello di software già presenti. Questo in modo da avere una distribuzione basata più sulla creatività e unicità che sulla quantità di app disponibili.
- Se l'APK è firmato, la chiave utilizzata per la firma deve scadere dopo il 22 ottobre 2033. Questo per garantire agli utenti dello Store di aggiornare le proprie applicazioni senza problemi.
- I dati sensibili devono essere gestiti in totale sicurezza.
- E' proibito dare presentazioni ingannevoli.
- I prodotti o gli annunci pubblicitari in essi contenuti non devono imitare le funzioni o le notifiche del sistema operativo o di altre applicazioni. Questo perché l'utente deve poter riconoscere una pubblicità dall'applicazione, altrimenti utilizzarla diventa frustrante e difficile.
- Gli annunci pubblicitari non devono essere invasivi, né limitare l'esperienza dell'utente.
- E' vietato riempire un certo elemento di parole chiave, così che compaia sempre nella pagina dei risultati della ricerca, anche se non ha rilevanza.

- Alcuni contenuti sono vietati, come ad esempio contenuti sessualmente espliciti, la violenza e l'incitamento all'odio.

Su internet è possibile trovare sia il contratto per gli sviluppatori, sia la polizza sui contenuti.
All'indirizzo *https://play.google.com/about/developer-content-policy.html* troviamo il centro norme per gli sviluppatori, diviso in sezioni, ognuna riguardante uno degli aspetti che l'applicazione deve rispettare prima di essere pubblicata. Riportiamo alcune di queste:



DISTRIBUTING EMERJOLLY

Every programmer writes code for two reasons: because he needs it, or because he needs to get something from it, be it money, success, or personal satisfaction.
Once you've finished writing an application, you must decide how to distribute it.
Software can be:

1. **Open Source**
   The legal owner of the copyright makes the code public, thus giving everyone the possibility to read, copy, edit and improve the code. Generally these types of software are free, but it's not a rule. The developer could also add the possibility to make voluntary donations.
   The primary purpose of open source licenses is the **survival of the software**, and its improvement.
   The biggest example of Open Source software is Linux: an operating system made of a software collection.

2. **Freeware**
   The software is public but the code is not: everyone can download and use the program completely for free, but nobody has access to the code. There can be limitations regarding commercial use, depending on what the developer prefers.

3. **Adware**
   Adware software is very much like freeware, with only one difference: the developer includes advertisement in his program, and gets some money from the company he advertises. The ads can be small and simple, or big and intrusive. They could even be *spywares*: small software that collects personal data without the user's permission, and sends it to societies that use it for statistics, or for commercial purposes. The problem is not the collection of the data itself, but the fact that the user doesn't know it's happening: programs like Google's ToolBar also collect personal data, but they ask for the user's permission first.

4. **Shareware**
   A shareware program is a program that can be used for free for a certain period of time, usually called "trial". Once the trial is over, the program must be purchased.
   The trial version may not include a lot of functions, which can be used only if you buy the program, but that's not always the case.

5. **Liteware**
   Liteware programs offer only a limited amount of functions for free but, unlike shareware software, there's no limit on time: a user can use only the free functions of the program for years, and never buy the complete version.

We want our application to be **freeware**. Since we've developed it for emergency situations, we don't think it's right to make people pay in order to benefit from it. We don't want the code to be public, because we've put a lot of effort into it and we want to keep it to ourselves.
We've thought about adding the possibility of voluntary donations, while we absolutely don't want to make money from advertisement, because we find it annoying.

SITOGRAFIA

- Android Developer
  https://developer.android.com/develop/index.html
- Stack Overflow
  http://stackoverflow.com/
- Centro norme per gli sviluppatori
  https://play.google.com/about/developer-content-policy.html
- Contratto di distribuzione per gli sviluppatori Google Play
  https://play.google.com/intl/ALL_it/about/developer-distribution-agreement.html