Accessing Preferences and Databases in Android

Michaël Van Canneyt

February 27, 2015

Abstract

A previous article we've shown how to construct a basic Android application using Free Pascal and the Java API. In this article, we'll show how to access the Preferences API and how to create and access a database under Android, as well as showing how data-bound controls work in the Android API.

1 Introduction

Armed with the techniques of the previous contribution about the Android API, it is possible to take further steps in the Android world. In this contribution, we'll explore 2 important parts of the Android API:

Preference management Android offers a rich API for fetching and setting preferences. It offers the possibility to display a preferences dialog, without having to write a single line of code.

Database management Every Android device has the sqlite engine installed. Applications can create and access an arbitray number of sqlite databases stored on the device using the Android API. Controls can be bound to the result of queries on this sqlite database, making it easy to create data-aware applications.

To demonstrate this, a small application is built: an application to register the presence or absence of students (pupils) in a school (we'll call it 'absenteeapp'). The use scenario is a teacher or administrative aid who does a round of the school buildings, visits all classrooms, and on his Android device notes who is absent in each class: he or she selects a group (by name), and gets a dialog which contains all students that should - theoretically - be present in this group. She or he ticks off whoever is absent. At the end of the round, the data can be transferred to some kind of administrative software.

An alternative use case is that all teachers can - on their own Android device - simply note the absent students for the group they are teaching currently, and transfer the result for their classes whenever they feel like.

To be able to do this, we'll need a database with 4 tables:

Groups A table with the name of groups.

Pupil A table with the names of students (pupils).

PupilGroup A table which keeps track of which student is part of which group: we assume that a studentl can be part of more than one group.

presence a table that notes who is present today.

The use case where the administrative aid does a round of all school buildings and classrooms necessitates some order in the display of the groups: Ideally, the order in which the classrooms are visited. This can be modeled using a preference: for simplicity, we'll assume that there are 2 orders: ascending and descending. In reality, there would probably be one or two ordering fields in the database.

Likewise, some people prefer the names of students to be sorted on first name, others on last name. This can be set using a second preference. In reality, each student will maybe have a number assigned to him (this is certainly the case in Flanders, Belgium), so ordering on this number could also be an option.

2 Activities in the application

The application will have 3 activities:

TGroupsActivity This is the main activity of the application: a list activity that displays a list of groups. One of the groups must be selected to display the second activity: `TPupilsActivity`.

TPupilsActivity This is again a list activity which displays a list of students, with a checkbox next to each student. This activity is started whenever the user selects a group in the main activity. When the 'back' button is pressed, the activity exits, and control returns to the main activity.

TAbsenteePreferenceActivity This is the activity which will allow the user to set the preferences. It is invoked from the options menu of the main activity.

The following android manifest file describes these 3 activities:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="eu.blaisepascal.absenteeapp"
  android:versionCode="1"
  android:versionName="1.0">
  <uses-sdk
    android:minSdkVersion="9"
    android:targetSdkVersion="9" />
  <application android:label="@string/app_name"
               android:icon="@drawable/icon">
    <activity android:name=".TGroupActivity"
              android:label="@string/app_name">
      <intent-filter>
         <action
            android:name="android.intent.action.MAIN" />
         <category
            android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
    <activity android:name="TPupilsActivity"/>
    <activity android:name="TAbsenteePreferenceActivity"/>
  </application>
</manifest>
```

The format of this manifest file has been discussed in the previous article, it will not be repeated here.

3 A database back end

The list of groups and students in the groups and presence records are stored in an SQLite database on the device. This database must be created by the app if it does not yet exist: It cannot be stored in the resources of the application.

Luckily, Android offers an API which takes care of creating (and even updating) a database: This functionality is present in a class called `ADSSqliteOpenHelper`.

An application which needs to use SQLite databases should create a descendant of this class and override some methods:

onCreate this method must create an empty database by executing some DDL SQL statements.

OnUpgrade Each database created by the Android SQLite helper API gets a version number (an integer). If a database must be upgraded, this method is called. The current and new version are passed to this routine, and it should execute all SQL statements needed to update the database.

onOpen This method is called whenever the database is opened.

Create The constructor of this class. It must set the expected current database version. This version will be checked against the version of an existing database, and will call 'OnUpgrade' if they do not match.

For the absentee application, the helper class is called `TAbsenteeDBHelper`. It overrides the above 4 methods:

```
TAbsenteeDBHelper = class(ADSSqliteOpenHelper)
public
  constructor Create(aContext: ACContext);
  procedure onCreate(aDatabase: ADSSqliteDatabase); override;
  procedure onUpgrade(aDatabase: ADSSqliteDatabase;
                      aOldVersion,
                      aNewVersion: jint); override;
  procedure onOpen(aDatabase: ADSSqliteDatabase); override;
end;
```

Some of these methods have an `aDatabase` parameter of type `ADSSqliteDatabase`: This is because the `ADSSqliteOpenHelper` is a helper class used in database life cycle management that does not offer any real database functionality. Instead, the `ADSSqliteDatabase` class presents the actual interface to an SQLite database. It offers methods to run queries, and retrieve data using cursors.

When a new database must be created, the `OnCreate` method is called. It executes the necessary SQL statements to create the database. This is done using the `ADSSqliteDatabase` interface, which offers an `ExecSQL` Method. The `ExecSQL` method gets an SQL statement as a parameter, which it executes, much like TSQLQuery in FPC.

```
procedure TAbsenteeDBHelper.onCreate(aDatabase: ADSSqliteDatabase);

begin
```

```
  aDatabase.execSQL(CreateTableGroup);
  aDatabase.execSQL(CreateTablePupil);
  aDatabase.execSQL(CreateTablePupilGroup);
  aDatabase.execSQL(CreateTablepresence);
end;
```

SQL statements executed like this must not return a result. The 4 SQL statements that create the absentee database are specified in 4 string constants, which the interested reader can check in the sources.

Since the application needs some data, some sample data is inserted in the tables. In a real-world application, this data would be fetched from a central administrative system. Here we simply create some sample data:

```
// Create Groups
For I:=1 to 6 do
  begin
  n:=JLLong.Create(i).toString;
  Q:='insert into Group (gr_name) values (''Group '+n+'A'')';
  aDatabase.ExecSQL(Q);
  Q:='insert into Group (gr_name) values (''Group '+n+'B'')';
  aDatabase.ExecSQL(Q);
  end;
```

Pupils are created and linked to the groups with similar statements.

The other methods of the `ADSSqliteOpenHelper` subclass are empty, except the Create method:

```
constructor TAbsenteeDBHelper.Create(aContext: ACContext);
begin
  inherited Create(aContext,'absentees', Nil, 1);
end;
```

The second argument is the name of the database. No path should be specified: The Android API stores all databases in a default, application-specific, location. The last argument is the version.

The Android API will check for the existence of this database: if it does not exist, it will call onCreate to create it. If it exists, but has the wrong version, the 'onUpgrade' method will be called instead.

The `ExecSQL` call cannot be used to run a select query, since it cannot handle a result set. To handle a result set, the `rawQuery` method of the `ADSSqliteDatabase` interface must be used.

To get an instance of the `ADSSqliteDatabase` interface in a `TAbsenteeDBHelper` method, the `getWritableDatabase` method must be used: this ensures that an instance of `ADSSqliteDatabase` is returned which can write to the database. Using this we can now write a method that returns the names of all the groups, sorted either ascending or descending:

```
function TAbsenteeDBHelper.GetAllGroups(sortdesc : Boolean): ADCursor;
var
  query: String;
begin
  query := 'select gr_id as _id, gr_name from Groups order by gr_name ';
```

```
  if SortDesc then
    query := query + ' desc'
  else
    query := query + ' asc';
  Result := getWritableDatabase.rawQuery(query, Nil);
end;
```

Since the result is a read-only record set, we could have used `getReadableDatabase` as well: it returns an instance which is able to read from the database, but can not necessarily write to it.

The `TAbsenteeDBHelper` class has some other methods to retrieve a list of students and set the presence or absence of a pupil, but these methods are similar to the methods above.

4 The main activity

The Group activity is the main activity. It displays a simple list of group names (fetched from the database), from which one must be selected. Because such an activity is very common, Android offers a ready-made list activity class: `AAListActivity`. The groups activity will therefor be a descendent of this list activity. It is defined as follows:

```
TGroupActivity = class(AAListActivity,
  AWAdapterView.InnerOnItemClickListener)
protected
  FGroupsDescending : Boolean;
  FPupilsFirstName : Boolean;
  fAdapter: AWSimpleCursorAdapter;
  fDataHelper: TAbsenteeDBHelper;
  fSelectedID: jlong;
  procedure FillItems(isRefresh: Boolean);
public
  procedure onCreate(savedInstanceState: AOBundle); override;
  Function onCreateOptionsMenu(AMenu: AVMenu) :  boolean ; override;
  Function onOptionsItemSelected(AItem: AVMenuItem) : Boolean; override;
  procedure onItemClick(aParent: AWAdapterView;
                        aView: AVView;
                        aPosition: jint; aID: jlong);
end;
```

The declaration contains 2 booleans which we'll use to store the preference values. The `AWAdapterView.InnerOnItemClickListener` interface is needed to respond to a click event from the list.

The Group Activity is also the main activity of the application, and the preferences dialog should also be launched from this. That means that we must create and populate the options menu. We've seen how to do this in the previous article:

```
function TGroupActivity.onCreateOptionsMenu(AMenu: AVMenu): boolean;

begin
  inherited;
  AMenu.add(0,AVMenu.FIRST,0,R.strings.Preferences);
  Result:=True;
end;
```

5 Getting preferences

Android supports saving and retrieving application preferences in a simple and straightforward way, using the PreferenceManager class. The default way is to save preferences in a standardly named file in a standard directory, using XML:

```
/data/data/eu.blaisepascal.absenteeapp/shared_prefs/
  eu.blaisepascal.absenteeapp_preferences.xml
```

The use of XML is hidden in the API, all the user of the API needs to know is how to read or write settings.

The PreferenceManager class has a function `getDefaultSharedPreferences` which returns an interface of type `ACSharedPreferences` that can be used to manage the preferences. It has the functions one would expect from an API to manage preferences:

```
function getString(para1: JLString; para2: JLString): JLString;
function getStringSet(para1: JLString; para2: JUSet): JUSet;
function getInt(para1: JLString; para2: jint): jint;
function getLong(para1: JLString; para2: jlong): jlong;
function getFloat(para1: JLString; para2: jfloat): jfloat;
function getBoolean(para1: JLString; para2: jboolean): jboolean;
function contains(para1: JLString): jboolean;
```

These calls are not very special, in fact they resemble the methods found in the `TIniFile` class.

6 Creating the Group activity

When the group activity starts, it must do 2 things: get the preferences, and connect to (or create) the database connection to get the group names. All this must be done in the `onCreate` method of the activity, which can look like this:

```
Const
  PrefGroups = 'GroupsDescending';
  PrefPupils = 'PupilsFirstName';

procedure TGroupActivity.onCreate(savedInstanceState: AOBundle);

Var
  Prefs : ACSharedPreferences;

begin
  inherited onCreate(savedInstanceState);
  setTitle(R.strings.name_groups);
  Prefs:=APPreferenceManager.getDefaultSharedPreferences(GetBaseContext());
  FGroupsDescending:=Prefs.getBoolean(PrefGroups,False);
  FPupilsFirstName:=Prefs.getBoolean(PrefPupils,False);
  fDataHelper := TAbsenteeDBHelper.Create(Self);
  FillItems(False);
  getListView.setOnItemClickListener(Self);
end;
```

The code starts by setting the title of the activity: since there is no resource file associated with the activity, it must be set manually. It then reads the preferences using the preference manager: this is done prior to creating the database. When the preferences have been read, the database helper is created: this will create and populate the database, if it didn't exist yet.

Last but not least, the list view is filled with items in the FillItems call, and the list InnerOnItemClickListener is set so the listview can react to clicks.

The `FillItems` method of the `TGroupActivity` call fills the listview with items.

A list in Android is - just as in Gtk or Qt, but unlike as in windows - just a container widget: it displays a list of other widgets, and simply scrolls these widgets on the display. Even simple text lists are rendered using `TextView` widgets. From this it follows that we must tell the list view which widgets it needs to load and display. This is done using a layout: each widget is created from a layout, and for our list of groups, this is the layout we'll use:

```
<?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:weightSum="100"
    android:orientation="horizontal" >
    <TextView
        android:id="@+id/group_name"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="30"
        android:padding="10dip"
        android:textSize="16sp" >
    </TextView>
</LinearLayout>
```

This layout describes 1 item in the listview. In the example above, it is a linear layout, containing 1 `TextView` widget. The values of the various attributes are mostly self-explaining, and most of them have been discussed in the previous article.

The widget has an ID associated with it: this is specified by the `android:id` attribute: The value `"@+id/group_name"` tells the resource packager that it must create an ID that will identify the widget instance - relative to its parent. The value of this ID is written to the resource file and can be used to designate the widget in code, we'll get back to this later.

A listview gets its items from a `ListAdapter` class. This is an abstract class from which descendants can be made. One of these descendants is the `AWArrayAdaptor` class, which simply takes an array of strings (or Java objects) and converts each item in the array to a listview widget. That corresponds to the default use of a stringlist in Delphi or Lazarus.

A second class, called `AWSimpleCursorAdapter`, takes a database query result cursor and creates an item for each record in the database query result: this is the data-aware version of the ListAdaptor.

For the Absentee application, this means a query must be run that returns all groups: Such a cursor is created in `GetAllGroups` in the database helper class. This cursor can be used to fill the listview with the names of the groups with the following code:

```
procedure TGroupActivity.FillItems(isRefresh: Boolean);
```

```
var
  cur: ADCursor;
  colfrom: array[0..0] of JLString;

begin
  if Not isRefresh then
    begin
    cur := fDataHelper.GetAllGroups(FGroupsDescending);
    startManagingCursor(cur);
    colfrom[0]:=TAbsenteeDBHelper.ColumnGroupName;
    fAdapter:=AWSimpleCursorAdapter.Create(Self,
                R.layout.group_list_item, cur,
                colfrom, [R.id.group_name]);
    setListAdapter(fAdapter);
    end
  else
    fAdapter.getCursor.reQuery;
end;
```

If the list is not being refreshed, then the cursor is created: it is the select query presented earlier, which selects all the groups in the groups database. The `StartManagingCursor` tells the activity that as soon as it stops, the query should be deactivated. When the cursor is fetched, the adaptor `AWSimpleCursorAdapter` is created. The constructor needs several arguments:

- The layout to use when creating an item. The layout is specified using its ID.
- The cursor, this is the cursor created with GetAllGroups.
- an array of field names to use.
- an array with ids of the items to fill with the fields specified in the preceding argument.

In our example, there is only 1 item which must be filled with the content of the group name field.

Once the adaptor is created, it is assigned to the list view. The result looks as in figure 1 on page 9.

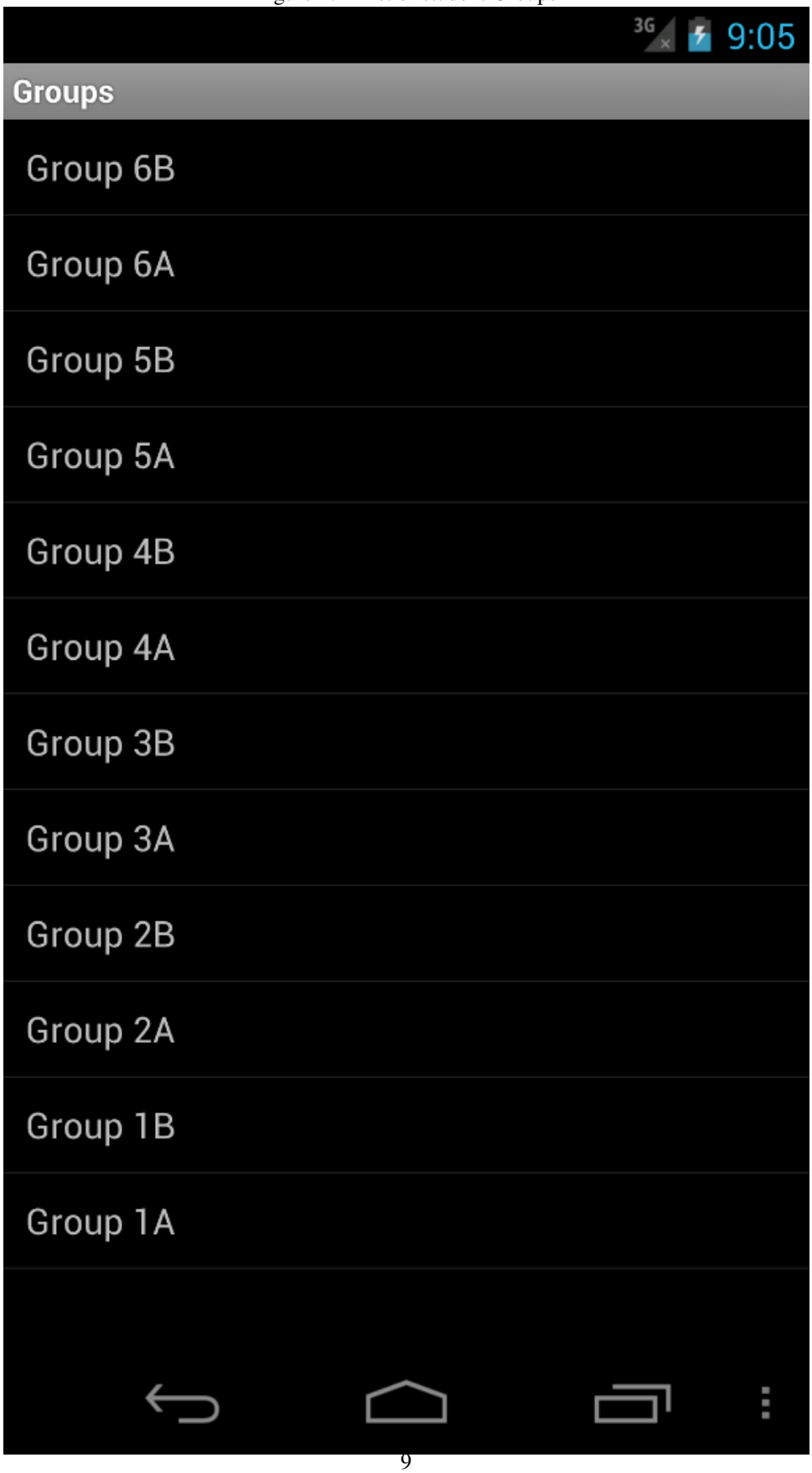
7 Displaying the students and noting absentees

The `TGroupActivity` instance was set as the item click listener of the listview. This means that the `onItemClick` method of the TGroupActivity instance will be called whenever the user clicks an item. The idea is that when a group is clicked, the list of students is shown.

Opening the list of students, can be coded as follows:

```
procedure TGroupActivity.onItemClick(aParent: AWAdapterView;
                                     aView: AVView;
                                     aPosition: jint;
                                     aID: jlong);
var
  intent: ACIntent;
```

Figure 1: A list of student Groups



```
begin
  intent := ACIntent.Create(Self, JLClass(TPupilsActivity));
  intent.putExtra('eu.blaisepascal.absenteeapp.ID', aID);
  intent.putExtra('eu.blaisepascal.absenteeapp.SortFirstName',FPupilsFirstName);
  startActivity(intent);
end;
```

The intent constructor needs 2 parameter: a context, and an Activity class to start. The context is the group itself, and the class is obviously the pupils activity.

An intent can carry parameters, which are simply named values: These parameters are then made available in the receiver of the intent. For the students dialog, one parameter to pass is obviously the ID of the group for which the students must be shown. A second (boolean) parameter is the preference setting for the sort order of the students.

The list of students is similar to the list of groups, except that there should be a check box for each pupil, to mark whether the student is present or absent. To achieve this, the layout of the groups listview can be copied, but must be enhanced with the following entry after the `TextView` tag:

```
<CheckBox
    android:id="@+id/present"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentRight="true"
    android:layout_marginLeft="4px"
    android:layout_marginRight="10px" >
</CheckBox>
```

The changed layout can be saved as pupil_list_item.xml. The `id` attribute shows that the check box will have an ID assigned to it, called `present`

The code for the students (pupils) listview activity is almost identical to the one for the groups. There are some small differences: for instance, in the `onCreate` method, the extra parameters passed with the intent must be retrieved, so it can be used in the query to retrieve all students in the group:

```
procedure TPupilsActivity.onCreate(savedInstanceState: AOBundle);

Const
  SPrefSort = 'eu.blaisepascal.absenteeapp.SortFirstName';
  SParamID = 'eu.blaisepascal.absenteeapp.ID';

begin
  inherited onCreate(savedInstanceState);
  fDataHelper:=TAbsenteeDBHelper.Create(Self);
  fSortFirstName:=getIntent.getBooleanExtra(SPrefSort,False);
  fID:=getIntent.getLongExtra(SParamID, 0);
  if (fID=0) then
    Raise EAbsenteeData.Create('No ID for absentee app given');
  SetTitle(R.strings.name_pupils);
  FillItems(False);
end;
```

The `Intent` instance can be retrieved with `getIntent`, it offers several methods to retrieve extra data in a variety of data types. Note that, together with the ID of the student group, the preference for sorting the students is also fetched.

The code to fill the list of items is virtually identical to the one for groups, with the difference that a different query is used: There are 2 fields: one for the name, one for the absence indicator. To respond to the user checking or unchecking a name, a custom binder is used:

```
procedure TPupilsActivity.FillItems(Isupdate: Boolean);

var
  cur : ADCursor;
  colfrom : array[0..1] of JLString;
  b : TCheckBoxListBinder;

begin
  if IsUpdate then
    fAdapter.getCursor.requery
  else
    begin
    cur:=fDataHelper.GetPupilPresenceFromGroup(FID,fSortFirstName);
    startManagingCursor(cur);
    colfrom[0] := 'pu_name';
    colfrom[1] := 'pr_code';
    fAdapter := AWSimpleCursorAdapter.Create(Self,
                  R.layout.pupil_list_item, cur, colfrom,
                  [R.id.pupil_name,R.id.present]);
    b:=TCheckBoxListBinder.Create;
    b.flistener:=Self;
    fadapter.setViewBinder(b);
    setListAdapter(fAdapter);
    end;
end;
```

The custom binder class is used to format each item in the list with the data from the list, for this it implements the `AWSimpleCursorAdapter.InnerViewBinder` class.

```
TCheckBoxListBinder = Class(JLObject,
                            AWSimpleCursorAdapter.InnerViewBinder)
  FListener : AWCompoundButton.InnerOnCheckedChangeListener;
  Function setViewValue(view: AVView;
                        cursor: ADCursor;
                        colindex : jint) : jboolean;
end;
```

the `FListener` field is an interface that is passed to this object: the binder will pass this interface to each checkbox it formats: the checkbox will call this interface when the checkbox is checked or unchecked..

The `setViewValue` method of the interface, which does the actual formatting (or binding) is implemented as follows:

```
function TCheckBoxListBinder.setViewValue(view: AVView;
  cursor: ADCursor;
  colindex: jint): jboolean;

Var
  cb : AWCheckbox;
  puID,i : jint;
```

```
  s : string;

begin
  Result:=(colIndex=2);
  if Result then
    begin
    cb:=AWCheckBox(view);
    puID:=cursor.getint(0);
    cb.setTag(TIDTag.Create(puID));
    i:=cursor.getColumnIndex('pr_code');
    if not Cursor.IsNull(i) then
      begin
      s:=cursor.getstring(i);
      cb.setChecked(s='+');
      end;
    cb.setOnCheckedChangeListener(flistener);
    end;
end;
```

The code is pretty straightforward: if the passed view is the checkbox – this can be determined from the column index – then it gets a tag associated with it. The tag can be any object. In this case a small custom defined object that just contains the ID of the pupil: this will be used in the `OnCheckedChange` listener. After setting the tag, the check marker is set or cleared, based on the value of the `pr_code` field (a '+' or '-').

Finally, the `OnCheckedChange` listener of the checkbox is set to the interface in FListener.

This `OnCheckedChange` listener interface is implemented in the `TPupilsActivity` class. It starts by getting the tag value which was added during the bind operation: it contains the students ID, and uses this to save the value of the check to the database using the data helper class:

```
procedure TPupilsActivity.onCheckedChanged(cb: AWCompoundButton;
  checked: jboolean);

Var
  puID : Integer;

begin
  puID:=TIDTag(cb.getTag()).ID;
  FDataHelper.Pupilpresence[puID]:=cb.isChecked;
end;
```

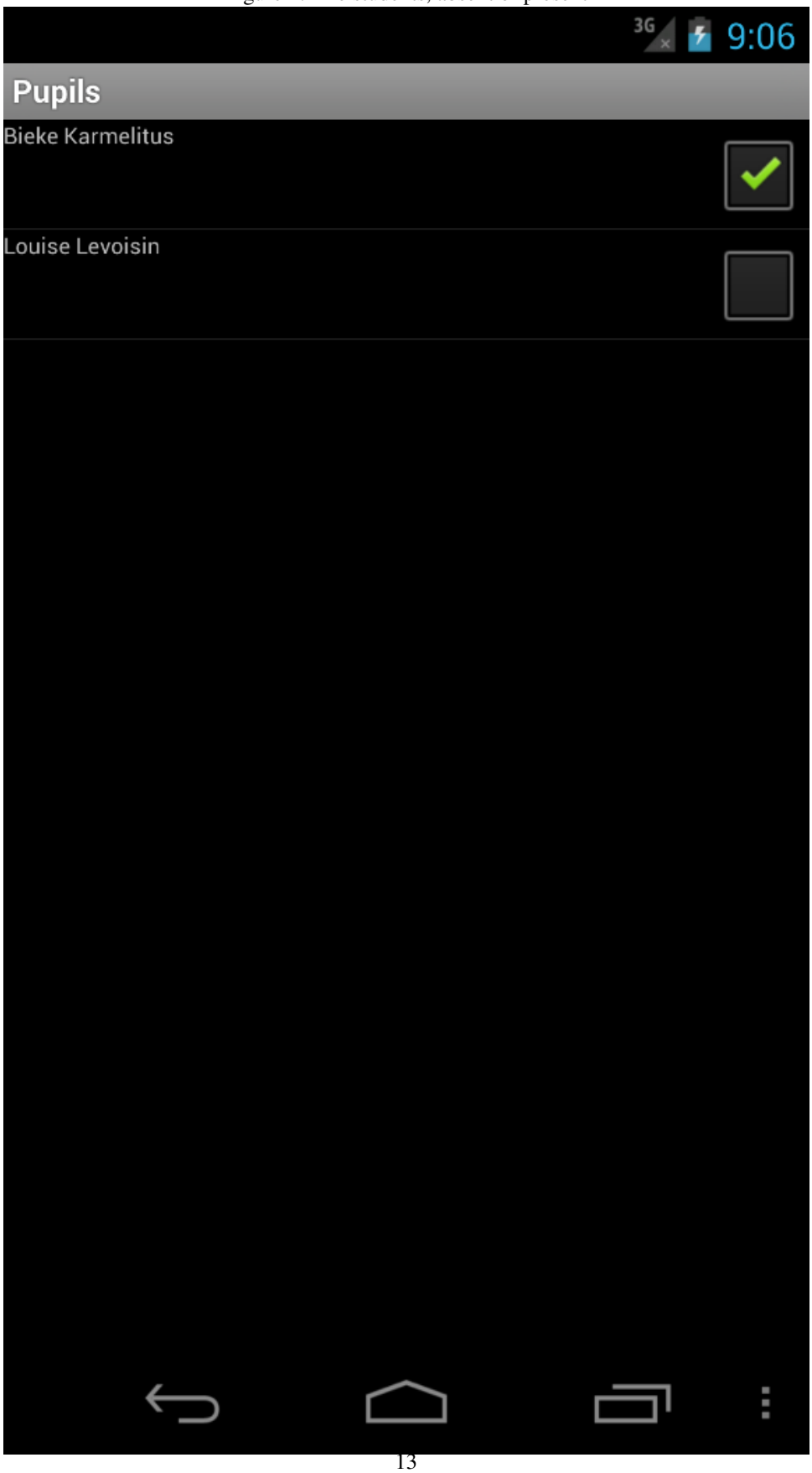
The actual queries will not be presented, they are straightforward and simple.

The activity, once called from the main groups activity, looks as in figure 2 on page 13.

8 Managing preferences

Besides an API for getting and setting preferences, Android offers also an API to create a dialog in which the user can manipulate the preferences: it offers a `PreferenceActivity` class which shows a dialog to manage the preferences of an application. This class can be used as-is, and does not need any additional code to manage the settings: it must be con-

Figure 2: The students, absent or present



structed using a special set of views, each of which is connected to a preference setting, using a name.

All that needs to be done, is specify a layout:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory
    android:summary="View settings"
    android:title="GUI Settings" >
  <CheckBoxPreference
     android:key="GroupsDescending"
     android:summary="When checked, groups are sorted descending"
     android:defaultValue="false"
     android:title="Sort groups descending" />
  <CheckBoxPreference
     android:key="PupilsFirstName"
     android:summary="When checked, pupils are sorted on firstname"
     android:defaultValue="false"
     android:title="Sort pupils on first name" />
  </PreferenceCategory>
</PreferenceScreen>
```

The layout is divided in categories (for simplicity there is only 1 category : “Gui setting”), which can have a title and summary. These act as a purely visual divider of the activity.

In each category, several preference widgets can be shown. There are different preference widgets, depending on the kind of preference that must be managed. Android offers out of the box a `CheckboxPreference` (or a `SwitchPreference`) for boolean values, `EditTextPreference`, `ListPreference`, `MultiselectListPreference`: the names speak for themselves.

Each preference has a title and summary, and a default value. The `key` attribute gives the name of the preference, which should of course match the name used to retrieve the preference in code.

All that is needed to show the above preferences, is to declare the activity class, and load the preferences resource in the `onCreate` method:

```
TAbsenteePreferenceActivity = class(APPreferenceActivity)
public
  procedure onCreate(savedInstanceState: AOBundle); override;
end;

Procedure TAbsenteePreferenceActivity.onCreate(savedInstanceState: AOBundle);

begin
  inherited onCreate(savedInstanceState);
  addPreferencesFromResource(R.layout.prefs);
end;
```

That is all there is to creating a (simple) preferences dialog.

The preferences activity must be shown when the user selects the ’Preferences’ item in the options menu, which means it must be implemented in the `onOptionsItemSelected` method of the group activity.

Showing the preferences dialog is done in the same way as showing any other activity, using an intent:

```
function TGroupActivity.onOptionsItemSelected(AItem: AVMenuItem): Boolean;

var
  intent: ACIntent;

begin
  if AItem.getItemId()=AVMenu.First then
    begin
    intent := ACIntent.Create(Self, JLClass(TAbsenteePreferenceActivity));
    startActivity(intent);
    Exit(True);
    end
  else
    Result:=Inherited onOptionsItemSelected(AItem);
end;
```

All in all, surprisingly litle code to create and show a preferences dialog. Android has more goodies when programming a preferences dialog, but it would lead too far to discuss them all in this article. in figure 3 on page 16, the dialog is shown in action.

9 conclusion

In this article, we've shown how to create a database backend for an Android application. Additionally we've shown how to set, store and retrieve preferences: both tasks are part of almost any application, and Android makes them particularly easy. In a future contribution we'll show how to connect to internet: a commonplace operation on a mobile device.

Figure 3: The preferences dialog