Using the Android SDK in FPC

Michaël Van Canneyt

February 26, 2012

Abstract

Android has a number of ways to program an android application. One of them is the NDK (Native Development Kit), which allows to create native code for the Android. The traditional way is to use the Java SDK. In this article, using the SDK from a Free Pascal program is explained.

1 Introduction

Android offers 2 ways to program applications for the Android platform. The NDK (Native Development kit) allows to execute native code in Android applications. The regular Android API is a Java API. Being a compiler, Free Pascal creates native binary code for the hardware Android runs on (the ARM CPU); therefor the NDK can be used by FPC, and how to leverage this native API with FPC has been explored in an article by Felipe Monteiro de Carvalho [Reference?].

However, Free Pascal can also create Java Byte code. How to do this using the Free Pascal Java back end, was explored in a previous article [Reference]. In this article we'll show how to use the Java back end to produce a GUI android application using the Java API.

To do this, a recent version of the FPC Java Back end compiler is needed. Information about the JVM (Java Virtual Machine) back end can be found in the Free Pascal wiki:

`http://wiki.freepascal.org/FPC_JVM`

It allows to download a ready-to use compiler for a select number of platforms. Instructions to compile the compiler for Java Byte code can be found on

`http://wiki.freepascal.org/FPC_JVM/Building`

A revision after revision 19830 is needed to be able to compile for the Android platform.

The demonstration application is intended for teachers, to mark pupils as being absent or present during the course. It will consist of 2 screens: a first screen to select a group of pupils, and a second, displaying all pupils in the group with a check mark to indicate that they are present. The results will be stored in a database - the idea being that this database can somehow be synchronized with an administrative system.

2 The Android API

The Android API is huge. It is not possible to describe all its possibilities in the scope of one article. To get a feel of all possibilities, the AndroidR14 unit - part of FPC's runtime -

imports all classes at the disposal of the Android programmer. It has over 60.000 lines of code. In this article, only a few classes from the API will be used.

In order to understand the SDK, some basic Android terminology is needed. One of the most fundamental terms is an Activity: this is roughly equivalent to a window or dialog in a GUI desktop application. Since the typical android device has only a small screen, an activity usually fills the whole screen, and the user is restricted to engage in only one activity at a time. The Java API has a class called `Activity`, which represents such an `Activity` or window. In Free Pascal the class is called `AAActivity`, the prefix A is used for all classes in the Android API. The Second A stands for App, as the class is part of the application namespace: the first letter of each namespace part is prepended to the Pascal class name.

An activity consists of one or more widgets: these are visual elements on the screen. A text, a check box, a text entry box, a list, or a button are but a few examples. Each of these is represented by a class in the API. Quite often, the activity is just a single widget which fills the screen. For instance a list: the `ListView` widget contains a list of widgets (often just text widgets) which can be scrolled and from which one item can be selected. Since this is an often-recurring pattern in Android programs, a `ListActivity` class is offered which is an Activity containing a screen-filling listview.

Very important when programming a GUI is the idea of an Intent. This is roughly equivalent to a windows message, but is more broader. Intents are a small data object (almost a record) that is passed around in the system. One intent is to start an activity i.e. open a new window. Another intent is to start a telephone call: there are many pre-defined intents in the Android API. An intent can be explicit: to open a well-defined window, or implicit: e.g. 'create a contact'. In the case of an implicit intent, the Android API will look for an activity that 'responds' to the intent in all the activities that it knows about, and will launch this activity. Activities that are designed to respond to such implicit intents must publish this somehow.

Besides visual classes, the Android API offers many other classes. One of them is an interface to the SQLite database. Every android device contains the SQLite engine, and this can be used to store data on the device in standard ways. The Android API makes sure that the databases of an application are available to this application only, if the Android API is used. The Android interface to the SQLite database is coded in the `ADSSQLiteOpenHelper` class. This object can be used to create and update databases on the device: all that must be done is override the appropriate methods. It can also run a query and return a cursor which can be used to browse over the result: more about this later on.

3 Getting started

Now that some basic terminology has been made, let's see what needs to be done to create an Android application. First of all, the Android SDK must be installed. It contains many tools, some of which are indispensable to create an Android application. The SDK comes in many versions, 4.0.3 being the latest version at the time of writing, and can be downloaded from

`http://developer.android.com/sdk/index.html`

An android application must be created in a directory that contains at least an AndroidManifest.xml file. This file describes the application, and amongst other things tells the application engine which Java class will be instantiated when the application is started. It looks as follows:

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

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="de.freex.absentees"
  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"/>
   </application>
</manifest>
```

The file is pretty simple. There are only some interesting nodes and attributes. The `package` attribute gives the package name of the application. It must match the namespace used in the application (more about this later). The `application`'s `android:label` and `android:icon` contain the application display name and icon. The notation with the @ indicates that the actual values are contained in the application's resources: These are a list of XML or other files, which must be in well-known positions. Therefor, `@string/app_name` means to look in the strings resource, under 'app_name' the latter means to look in the icon resources for a file named 'icon'.

The manifest file must list the activities in the application: one `activity` tag per activity class.

The first activity name is specified as `.TGroupActivity`. The dot (.) means that the name should be prefixed with the package name, so the Android application layer will try to instantiate an Activity subclass called `de.freex.absentees.TGroupActivity`. The name of the activity hints at it's function: it will show the groups of pupils in the application.

The intent filter is used to indicate that the `TGroupActivity` class responds implicitly to the pre-defined intent `android.intent.action.MAIN` from the category `android.intent.category.LAUN`
This pre-defined intent starts the main activity of any android application.

The second activity in the application is the `TPupilsActivity`, it will be used to mark pupils as absent or present.

All labels are put in a strings resource, and referred to by name.

Android resources must be put in specific places. All resources must be in a directory `res`, with some subdirectories. The layout of the resource directory is described in detail on

`http://developer.android.com/guide/topics/resources/providing-resources.html`

The string values must be stored in a file strings.xml in the values directory. It looks as follows:

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

```
<resources>
    <string name="app_name">FreeX Absentees</string>
    <string name="name_Groups">Groups</string>
    <string name="name_Pupils">Pupils</string>
</resources>
```

The syntax is self-explaining.

Once all resources have been defined, they must be compiled into a format that the android API can read. This is done with a tool in the Android SDK, `aapt` (Android Asset Packing Tool):

```
aapt p -f -M AndroidManifest.xml -F bin/absenteeapp.ap_ \
     -I /path/to/android.jar -S res -m -J gen
```

There are many options to be learned:

p option tells `aapt` to create a package.

-f forces it to overwrite an existing file.

-M specifies the manifest file.

-F specifies the output file.

-I includes the android API jar in the package.

-S Specifies the subdirectory containing the resources. It will be scanned recursively.

-m Create package directories in the directory specified by `-J`.

-J Where to write the Java file containing the resource constant definitions.

Each resource in an Android application gets a numerical ID. The `aapt` tool generates a java file that contains a constant for each resource. It looks partially like this:

```
package de.freex.absenteeapp;

public final class R {
    public static final class id {
        public static final int group_name=0x7f050000;
        public static final int present=0x7f050002;
        public static final int pupil_name=0x7f050001;
    }
    public static final class layout {
        public static final int group_list_item=0x7f030000;
        public static final int pupil_list_item=0x7f030001;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
        public static final int name_Groups=0x7f040001;
        public static final int name_Pupils=0x7f040002;
    }
}
```

This class must be translated to pascal to be able to use it in pascal code. A rather straight translation would be:

```
unit Resources;
{$mode objfpc}{$H+}
{$namespace de.freex.absenteeapp}
interface

type
  R = class sealed
  public
  type
    id = class sealed
    public const
     group_name = $7f050000;
     present    = $7f050002;
     pupil_name = $7f050001;
    end;

    layout = class sealed
    public const
      group_list_item = $7f030000;
      pupil_list_item = $7f030001;
    end;

    strings = class sealed
    public const
      app_name    = $7f040000;
      name_Groups = $7f040001;
      name_Pupils = $7f040002;
    end;
  end;
```

The `app_name` resource ID can then be specified as `R.strings.app_name` in the pascal sources.

4 A database back end

The list of groups and pupils 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.

Android offers an API which takes care of updating or creating a database: all operations needed when working with databases. All this functionality is in a class called `ADSSqliteOpenHelper`. An application wishing to use SQLite databases must 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 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.

For the Absentee application, 4 tables must be created:

Group The list of groups. It contains an gr_id (numerical) and gr_name (string) field.

Pupils The list of pupils It contains an pu_id (numerical) and pu_name (string) field.

PupilGroup The connection between pupils and the groups they are in.

Presence this table contains a record per pupil per day, and a code denoting the presence: + for present, - for absent.

For the absentee application, the helper class is called `TAbsenteeDBHelper`. As a start, the following methods must be implemented:

```
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;
```

As can be seen, some of these methods receive an `aDatabase` parameter of type `ADSSqliteDatabase`. The `ADSSqliteOpenHelper` is just a helper class used in database life cycle management. It does not offer actual database functionality. The `ADSSqliteDatabase` class presents a low-level interface to the SQLite database. It offers methods to run queries, retrieve data using cursors.

The OnCreate method is called when a database must be created. It simply executes some queries to create the tables:

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

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

The `ExecSQL` method gets an SQL statement as a parameter, which it executes. The SQL statement should not return a result; in the above example, the SQL statements are specified in 4 string constants.

In a real-world application, this would be sufficient. However, for test purposes, some fake data is inserted in the tables. For instance, a list of groups is inserted with the `ExecSQL` method:

```
// 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;
```

Similar loops exist to create pupils and link them to the groups.

The other database helper methods 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 then check for the existence of this database, and check whether it has the correct version, and call the appropriate methods.

Below, some methods will be added to this object to query the absentee database.

5 The first activity

Now that we can access a database, it is time to create the first activity for the application. It should present a list of groups, and as soon as the user selects a group, the second activity must be opened.

The Android Manifest file specifies the name of the activity: `de.freex.absentees.TGroupActivity`. This is the name of the Activity class descendant that will be instantiated. Since the activity will only show a list, it is logical that the class should descend from `AAListActivity`, which is a pre-defined listview activity in the Android API.

As all activities in our application will start with `de.freex.absentees`, we'll set the namespace for classes to this value, using the `$Namespace` directive. Since all strings in a java application are unicode (UTF-16), the compiler can be told to use unicode strings by default, as this is more efficient. This is done using a `$modeswitch unicodestrings` directive:

```
Unit AbsenteeApp;

{$mode objfpc}{$H+}
{$namespace de.freex.absenteeapp}
{$modeswitch unicodestrings}

interface

Type
  { TGroupActivity }
  TGroupActivity=class(AAListActivity,
                       AWAdapterView.InnerOnItemClickListener)
  protected
    fAdapter: AWSimpleCursorAdapter;
    fDataHelper: TAbsenteeDBHelper;
    fSelectedID: jlong;
    procedure FillItems(isRefresh: Boolean);
  public
    procedure onCreate(savedInstanceState: AOBundle); override;
    procedure onItemClick(aParent: AWAdapterView;
                          aView: AVView;
                          aPosition: jint; aID: jlong);
  end;
```

As can be seen, the activity class is rather simple. It descends from `AAListActivity` and implements the `AWAdapterView.InnerOnItemClickListener` interface, which contains the `OnItemClick` method, which will be explained below.

The `onCreate` method is called when the activity is first created. In this method, the connection to the database must be made, as this is where the list of items to be displayed must be fetched from. This is done by creating an instance of the database helper `TAbsenteeDBHelper` and storing it for future reference.

```
procedure TGroupActivity.onCreate(savedInstanceState: AOBundle);
begin
  inherited onCreate(savedInstanceState);
  fDataHelper := TAbsenteeDBHelper.Create(Self);
  setTitle(R.strings.name_groups);
  FillItems(False);
  getListView.setOnItemClickListener(Self);
end;
```

After creating the database helper, the title is set from the string resources ID. The `UpdateData` call adds all items to the list. The last line sets the activity instance itself as the `OnItemClick` handler for the listview.

To fill the listview, the `FillItems` routine is used. A listview gets its items from a `ListAdapter` class. This is an abstract class from which descendants can be made. One of these descendants `AWSimpleCursorAdapter`, takes a database query result cursor and creates an item for each record in the database query result. 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. It can then be used to

```
procedure TGroupActivity.FillItems(isRefresh: Boolean);
var
  cur: ADCursor;
  colfrom: array[0..0] of JLString;
begin
  if Not isRefresh then
    begin
    cur:=fDataHelper.GetAllGroups;
    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 refreshed, then the cursor is created: it is a select query 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.

After this, a `AWSimpleCursorAdapter` instance is created and stored in `fAdapter`. It has the following constructor:

```
constructor create(Context: ACContext; Layout: jint;
```

```
                    Cursor: ADCursor;
                    const from: array of JLString;
                    const _to: array of jint)
```

The constructor needs a context (in our case, the activity) an ID of a layout resource to create for each item (more about this below) a cursor (the data from which to create the list) and then 2 arrays: an array of field names and an array of widget IDs. The widget IDs and the layout to be created are linked together in a layout file.

A layout file is used to specify the look of the application. It specifies the look and feel (layout) of the activity or the widgets contained in it. The layouts are XML files, which are packaged in the application's resources. At runtime, these layouts are converted by the android engine to a set of widget instances. Which widgets and how they are arranged on screen is described in the layout. For Lazarus or Delphi users, a layout is much like a form file. Each XML tag in the layout specifies a widget. The tag name equals the class name of the widget - this is how the Android layouter knows how to re-create the layout.

For a listview, a layout is used to create each item, even if it is just a textitem. A text item is drawn using a `TextView` widget, which can be specified in a layout file as follows:

```
<?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>
```

There are many attributes in a layout file: a complete description is beyond the possibilities of this article, but luckily the names are very descriptive. The point to remember is that the layout describes 1 item in the listview. In the example above, it is a linear layout, containing 1 `TextView` widget, indicated by the <>. The widget has an ID associated with it: this is specified by the `android:id` attribute. The other attributes of the textview tag control its appearance: the names and their values are pretty self-explaining.

The value `"@+id/group_name"` tells the resource packager that it must create an ID that will identify the widget instance - relative to it's parent. The value of this ID is written to the resource file and can be used to designate the widget in code, specifically when creating the cursor adapter:

```
fAdapter:=AWSimpleCursorAdapter.Create(Self,
                R.layout.group_list_item, cur,
                colfrom, [R.id.group_name]);
```

The `r.id.group_name` constant is the ID of the `TextView` widget, telling the adapter that the contents of the field `gr_name` (which is the only element in the `colfrom` array) should be applied to the `TextView` widget.

The above layout is saved in a file `res/layout/group_list_item.xml`. The resource packager will also create an ID for this resource and this is what is used when creating the Cursor Adapter.

The result of all this is a widget that displays a list of groups found in the database.

6 Creating and installing the app

The Java classes which are generated by Free Pascal's JVM backend must be converted to bytecode which is suitable for the Android engine. The engine needs all classes, including the classes which are part of FPC's Java back end, so these must be all put in the same directory below the application directory, for example bin/classes.

```
JVMRTL=/home/michael/FPC/jvmbackend/rtl/units/jvm-android
cp $(JVMRTL)/*.class bin/classes
cp -r ($JVMRTL)/org bin/classes
```

The conversion to Android bytecode is done using the `dx` converter, which comes with the Android SDK:

```
dx --dex --output=bin/classes.dex bin/classes
```

This tells the converter that all java classes below bin/classes must be converted to a file bin/classes.dex.

The resulting file, combined with the resources can now be transformed to an android package:

```
apkbuilder bin/absenteeapp-unsigned.apk \
  -u -z bin/absenteeapp.ap_ \
  -f bin/classes.dex
```

the apkbuilder tool is also part of the Android SDK.

The android device only allows to install signed jar files, which means that the package file must be signed. This must be done by the `jarsigner` tool, which comes with the Java SDK. The jarsigner needs a keystore file, which can be generated with the following `keytool` command:

```
keytool -genkey -v -keystore /home/michael/.androidkeys \
        -alias michael -keyalg RSA -validity 10000
```

With the keystore file in place, the package can now be signed:

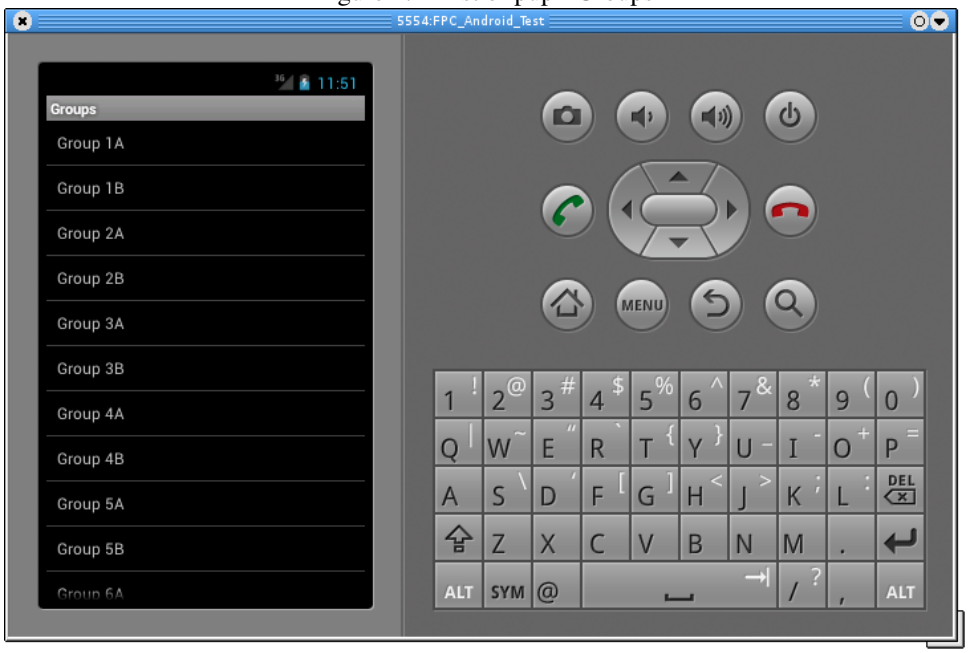
```
jarsigner -keystore /home/michael/.androidkeys \
          -signedjar bin/absenteeapp.apk \
          bin/absenteeapp-unsigned.apk michael
```

The command is pretty simple: it needs the output and input packages and an alias for the key to be used.

To install the app on the Android emulator, the emulator must be started, and the `adb` tool - part of the Android SDK - can be used to transfer the app to the running android instance:

```
adb -e install -r bin/absenteeapp.apk
```

Figure 1: A list of pupil Groups



The `-e` option tells the adb tool to install in the emulator. (to install on a device attached to the USB port, use the `-d` option instead) The `-r` tells the tool to reinstall, but keep all data associated with the app.

If all went well, the app should show up in the menu. When started, figure 1 on page 11 shows the groups in action in the Android emulator.

7 intents

When the user selects a group, a second screen should appear with the pupils of the group (`TPupilsActivity`).

This must be done using an `Intent` class. To open the `TPupilsActivity`, an explicit intent must be used: an explicit intent tells the Android engine that a specific activity must be started.

The `TGroupActivity` instance was set as the item click listener of the listview. This means that the `onItemClick` method of the group activity will be called whenever the user clicks an item. To open the list of pupils, it can be coded as follows:

```
procedure TGroupActivity.onItemClick(aParent: AWAdapterView;
                                     aView: AVView;
                                     aPosition: jint;
                                     aID: jlong);
var
  intent: ACIntent;
begin
  intent := ACIntent.Create(Self, JLClass(TPupilsActivity));
  intent.putExtra('de.freex.absenteeapp.ID', aID);
  startActivity(intent);
end;
```

The intent constructor needs 2 parameter: a context, and an Activity class.

The pupils activity will need to know which group to show the IDs from. This is determined by the `aID` parameter: When getting the list of groups from the database, it is mandatory that each record has a field called `_id`. The value of this field will be associated with the items in the listview.

The `_id` value is passed in the `onItemClick` method, and must be passed on to the new activity. With an intent, extra data can be associated. This is a list of named values, and they are added to the intent with the `putExtra` method. When the new activity is started, it can query the values associated with the intent that started it. In the above code, `aID` (the ID of the group) is passed on to the Intent instance as extra data.

8 Enhancing the listview

The list of pupils is similar to the list of groups, except that there should be a check box for each pupil, to mark whether the pupil is present or absent. To achieve this, the layout of the groups listview can be copied and 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 pupils listview activity is almost the same as the one for the groups. There are some minor differences. For instance, in the OnCreate method, the extra data passed with the intent is retrieved, so it can be used in the query to retrieve all pupils in the group:

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

begin
  inherited onCreate(savedInstanceState);
  fDataHelper := TAbsenteeDBHelper.Create(Self);
  fID := getIntent.getLongExtra('de.freex.absenteeapp.ID', 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.

The listview is filled in exactly the same manner as for pupils:

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

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

begin
  if not IsUpdate then
    begin
    // create cursor
    cur:=fDataHelper.GetPupilPresenceFromGroup(FID);
    startManagingCursor(cur);
    // columns
    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]);
    // Pass viewbinder:
    b:=TCheckBoxListBinder.create;
    b.flistener:=Self;
    fadapter.setViewBinder(b);
    // Set adapter.
    setListAdapter(fAdapter);
    end
  else
    fAdapter.getCursor.requery;
end;
```

The above code looks pretty similar to the one for the groups. There are 3 differences:

1. The list of pupils is fetched with a different cursor: The data helper's `GetPupilPresenceFromGroup` returns this cursor. Note that the ID value which was stored in the activity's constructor is passed to this method.
2. The `AWSimpleCursorAdapter` constructor gets 2 column names: one column name for the pupil's name (`pu_name`), and one for the contents of the check box (`pr_code`). Likewise, 2 widget ids are passed: `R.id.pupil_name`, `R.id.present`.
3. The adapter gets passed a ViewBinder instance with `setViewBinder`.

When the adapter lays out the items, it applies the data from the cursor to the widgets in the item. For this, it uses the fieldnames and widget IDs passed in the constructor. Sometimes, this mechanism is not powerful enough or does not know how to transform the field value to a usable value fot the widget. The Android API caters for this by allowing the programmer to pass a `ViewBinder` interface to the adapter. In the case of the checkbox, the viewbinder is needed to set the checkbox' `OnCheckedChange` listener: this is an event which is called when the check mark is set or cleared. It is also needed to convert the `pr_code` field (which is not a boolean field) to a boolean value for the `Checked` property of the checkbox.

The `TCheckBoxListBinder` class takes care of this. It is defined as follows:

```
  TCheckBoxListBinder = Class(JLObject,
      AWSimpleCursorAdapter.InnerViewBinder)
    FListener : AWCompoundButton.InnerOnCheckedChangeListener;
```

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

It is created in the above code, and the listener is set to the activity class itself:

```
  b:=TCheckBoxListBinder.create;
  b.flistener:=Self;
```

The viewbinder interface has only 1 method: `setViewValue`. It must handle the copying of data from the cursor to the widgets in the list item. If it did this, it should return `True`. If it returns `False`, the list adapter uses the default mechanisms to apply the field value to the widget. For the checkbox, the method 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
    // typecast checkbox
    cb:=AWCheckBox(view);
    // attach a tag to it with the pupil ID.
    puID:=cursor.getint(0);
    cb.setTag(TIDTag.Create(puID));
    // Set the checkbox Checked property.
    i:=cursor.getColumnIndex('pr_code');
    if not Cursor.IsNull(i) then
      begin
      s:=cursor.getstring(i);
      cb.setChecked(s='+');
      end;
    // set the listener
    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 is an object that just contains the ID of the pupil. It is 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. Finally, the `OnCheckedChange` listener is set.

The `OnCheckedChange` listener interface is implemented in the pupils activity class:

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

```
Var
  puID : Integer;

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

It retrieves the tag value with the pupil's ID, and saves this to the database using the data helper class. The interested reader can consult the sources to see which queries are executed.

The code is now again ready to be run. All the steps performed to create and install the app must be executed again. Now, when the app runs and a group is selected, the output should look as in figure 2 on page 16.

9 Conclusion

Creating an Android application with Free Pascal is not hard: the JVM backend of the compiler makes it easy to do. The largest problem by far is getting to know the Android API, which is very big. Once the basics have been mastered, setting up a makefile (or an Ant build environment) makes it easy to compile and deploy an Android to the android emulator or an android device.

Figure 2: A list of pupils