Using Free Pascal to create Android applications

Michaël Van Canneyt

January 19, 2015

Abstract

Since several years, Free pascal can be used to create Android applications. There are a number of ways to do this, each with it's advantages and disadvantages. In this article, one of the ways to program for Android is explored.

1 Introduction

Programming for Android using Free Pascal and Lazarus has been possible since many years, long before Delphi unveiled its Android module. However, it is not as simple a process as creating a traditional Desktop application, and to complicate matters, there are several ways to do it:

1. Using the Android NDK. Google makes the NDK available for tasks that are CPU-intensive (a typical example are opengl apps), but warns that this should not be the standard way of developing Android apps.

   This is the way Delphi made it's Android port: Using Firemonkey – which uses OpenGL to draw controls – this is the easiest approach.

   There is a Lazarus project that aims to do things in a similar manner (using the native-drawn widget set approach).

2. Using the Android SDK: a Java SDK. This is the Google recommended way to create Android applications.

This article shows how to create an Android application using the Java SDK.

2 FPC : A Java bytecode compiler

The Android SDK is a Java SDK: Android applications are usually written in Java. However, the Free Pascal compiler can compile a pascal program to Java Bytecode. This is done with the help of the JVM cross-compiler.

Free Pascal does not distribute this cross-compiler by default (yet), so it must be built from sources. The process is described in the WIKI:

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

It basically boils down to 3 things:

1. Install the Free Pascal sources. Using the subversion client:

```
svn co http://svn.freepascal.org/svn/fpc/trunk fpc
```

Windows users may find it easier to downlod the sources with the TortoiseSVN shell extension, the above URL will do just fine.

2. Install or compile jasmin. Jasmin is a tool that converts Java assembler (generated by the compiler) to Java bytecode. A patched version of Jasmin must be used to support some floating point constructs.

3. Build the cross-compiler. In a console (command) window, go to the top-level directory of the FPC sources, and execute the following command:

```
make CROSSOPT="-O2 -g" CPU_TARGET=jvm OS_TARGET=java all
```

For this to work, the `make`, `fpc` and `jasmin` compiler commands must be located in one of the directories in the PATH.

The above steps should leave you with a JVM compiler. You can install it with

```
make CROSSOPT="-O2 -g" CPU_TARGET=jvm OS_TARGET=java \
  INSTALL_PREFIX=/usr/local crossinstall
```

(Windows users should change the /usr/local with something like C:/FPC).

Once this is done, the new compiler can be tested with the following small program:

```
program hello;

uses jdk15;

begin
  jlsystem.fout.println('Hello, world !');
end.
```

Compile it:

```
ppcjvm hello.pp
```

You may need to add the path to the JVM rtl units on the command-line, something like `-Fu/usr/local/fpc/2.7.1/units/rtl-jvm` on unix-based operating systems. On Windows it will be something like `-Fuc:/FPC/units/rtl-jvm`.

Once that is done, the program is ready to run inside the Java virtual machine:

```
java -cp /usr/local/lib/fpc/2.7.1/units/jvm-java/rtl:. hello
```

The `-cp` option to the Java virtual machine tells it where it can find the system classes used by a FPC program.

Note that the JVM bytecode compiler does not have an RTL associated with it. This means that the sysutils unit is not available, the Lazarus LCL, synapse or indy are not available: these have not been ported (or cannot be ported) to Java Bytecode.

Instead, the complete Java SDK is at the disposal of the programmer. Java classes can be imported in Free Pascal using the `javapp` tool: this will convert Java classes to equivalent Pascal headers. It is a java tool, which is available in the Free Pascal source tree.

Specifically for Android, the Android SDK has also been imported using this tool.

3 Developing for Android

The Android platform is essentially a Java Runtime engine (Dalvik) running on top of a Linux kernel. Originally for the ARM processor, it has been extended to include the Intel platform as well. The Java Runtime offers a huge set of classes to handle everyday tasks on the Android platform: the Android API. It is beyond the scope of this article to cover the whole of the Android API.

Instead, the steps needed to create a Android application in Java Bytecode, generated from Pascal sources, will be outlined. The steps will be demonstrated by re-creating the SDK's sample 'Skeleton Application' in Object pascal. This is a simple application that has an edit control, a config menu and 2 buttons. One button exits the program, the other clears the edit control text. The options menu items perform the same action.

Several steps are involved in the process:

1. Installing the Android SDK and (optionally) create some android virtual devices in an emulator.
2. Creating some resource files (although this can be skipped, all layouts can be created in code if so desired).
3. Creating some java classes from Pascal code.
4. Creating an Android Application Manifest file.
5. Packaging all in an Android Package (.apk) File.

The first step is downloading and installing the Android API. It can be downloaded from

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

It is not necessary to install Android Studio: this is only useful if you want to develop in Java. The SDK can be downloaded separately from the same page. Once the SDK is downloaded, the android application (in the tools directory) must be started, and several extra packs must be downloaded separately. This can take some time: All versions of the Android API can be downloaded, but for testing, the latest version can be used.

Once this is done, the apkbuilder script (or batch file) must be downloaded: this script exists by default in older versions of the Android API, but not in newer versions (it is deprecated). It can still be found as a separate download e.g. on github.

4 Creating a manifest file

It is good practice to create a separate directory for each Android Application that must be developed. In this directory, a manifest file must be created, since every Android application needs a manifest file. By convention, this file is called AndroidManifest.xml.

The manifest file describes the application to the Android system, and amongst other things tells Android's runtime engine which Java class must be instantiated when the application is started. For the skeleton application, it looks as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="eu.blaisepascal.skeletonapp">
```

```
  <uses-sdk android:targetSdkVersion="14"/>
  <application
    android:label="@string/skeletonapp"
    android:icon="@drawable/icon">>
    <activity android:name=".TSkeletonActivity">
      <intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </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: this is an internal, unique name for the application.

This name must match the namespace used in the application's Java bytecode (more about this later).

The `application` tag contains 2 important attributes:

label this contains the display name of the application.

icon this contains the location of the icon to use for the application.

Note that the values for these attributes start with the @ symbol. This indicates that the actual values are contained in the application's resources: The application's resources (similar to windows resources) are a list of XML or other files, which must be in well-known locations.

Therefor, `@string/skeletonapp` means that the value is in the strings resource, under 'skeletonapp'. Likewise, the application's icon location is in the drawable folder – a general name for all images – and the icon image file is called 'icon'. (the extension is determined automatically)

The `activity` tag indicates which activities the android application contains. There is one tag per activity in the application. For each activity, some intents can be specified: an intent is an indication of the purpose of an activity. Activities will be explained in the following sections.

The main thing to remember here is that the `android:name` tag specifies the name of the Object Pascal class that must be instantiated when the application is started. The name starts with a dot (.), which means that the name is relative to the package name. In the case of the above manifest file this means that the full name of the class to start is:

```
eu.blaisepascal.skeletonapp.TSkeletonActivity
```

5 Activities and intents

In order to understand the Android programming model, some basic Android terminology is needed.

One of the most fundamental concepts when programming Android is the Activity: Roughly equivalent to a window or dialog in a GUI desktop application, this is the basis of an Android GUI application.

An activity usually fills the whole screen (device screens are small), so the user is restricted to interact with only one activity at a time. The Java API has a class called `Android.App.Activity`, which represents such an `Activity` or window.

In Free Pascal the full Android class name is abbreviated to `AAActivity`, where the 2 first parts of the class name (the namespace) have been abbreviated to AA: the first A is short for Android, the Second A stands for App.

An Android Activity consists of one or more widgets: these are visual elements on the screen. By itself, Android offers only a few widgets: A text, a check box, a text entry box, a list, or a button. Each of these is represented by a class in the API.

Because screen space is limited, an activity often consists of 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 selecting an item is an activity which occurs often in Android programs, a `ListActivity` class is offered which is an Activity containing a screen-filling ListView.

A second concept when programming an Android GUI is *Intent*. This important concept 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 such intent is to start an activity, i.e., open a new window.

Another intent can be to start a telephone call, send an SMS, etc.

There are many pre-defined intents in the Android API. An intent can be explicit: to open a well-defined window. But it can be 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.

Obviously, activities that are designed to respond to such implicit intents must publish this somehow: this is normally done in the manifest file.

The manifest file of the skeleton application indicates one Activity (`TSkeletonActivity`) and the intent filter is used to indicate that the `TSkeletonActivity` class responds implicitly to the pre-defined intent `android.intent.action.MAIN` from the category `android.intent.category.LAUNCHER`. This pre-defined intent starts the main activity of any Android application.

6 Resources

Android resources (or Assets) can contain many resources:

1. Images (called drawables)
2. Strings (texts)
3. Layouts
4. Colors
5. Styles

Most of these must be specified in XML files, but images are put in a file as-is. All resources must be in a directory (called `res`), with some subdirectories. The detailed layout and allowed files for the resource directory is described in detail on

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

In the manifest file, the application name was specified as a resource, a string resource. 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="skeletonapp">Pascal Skeleton App</string>
<string name="Back">Back</string>
<string name="Clear">Clear</string>
<string name="Hello">Hello World!</string>
</resources>
```

The syntax is self-explaining: each string value is in its own tag, and the `name` attribute gives the unique name for the string value: this is quite similar to Free Pascal resource strings.

The same can be done for other values such as colors:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="Red">#ff0000</color>
  <drawable name="SemiBlack">#80000000</drawable>
</resources>
```

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 called `aapt`, which is short for "Android Asset Packing Tool".

It is a command-line tool, and is invoked as follows:

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

There are many options to be learned:

-p This option tells `aapt` to create a package.

-f forces it to overwrite an existing file.

-M specifies the name of the manifest file.

-F specifies the output file.

-I includes the android API jar in the package.

-S Specifies the subdirectory containing the resources (res, in our case). 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.

When the asset packaging tool does its work, it will assign a numerical ID to each resource which it finds. This ID can then be used to load the resource from application code. The `aapt` tool generates a Java file that contains a numerical constant for each resource with the same name as used in the resource file. For the skeleton application, the file looks like this:

```
package eu.blaisepascal.skeletonapp;

public final class R {
  public static final class color {
    public static final int Red=0x7f040000;
  }
  public static final class drawable {
    public static final int SemiBlack=0x7f020002;
    public static final int icon=0x7f020000;
    public static final int violet=0x7f020001;
  }
  public static final class id {
    public static final int back=0x7f070001;
    public static final int clear=0x7f070003;
    public static final int editor=0x7f070000;
    public static final int image=0x7f070002;
  }
  public static final class layout {
    public static final int skeleton_activity=0x7f030000;
  }
  public static final class string {
    public static final int Back=0x7f050001;
    public static final int Clear=0x7f050002;
    public static final int Hello=0x7f050003;
    public static final int skeletonapp=0x7f050000;
  }
  public static final class style {
    public static final int ActionButton=0x7f060000;
    public static final int TextAppearance=0x7f060001;
    public static final int TextAppearance_ActionButton=0x7f060002;
  }
}
```

This file can be translated to Object Pascal, so the constants can be used in Object Pascal code:

```
unit Resources;

{$mode objfpc}{$H+}
{$namespace eu.blaisepascal.skeletonapp}

interface

type
  R = class sealed public
  type
    drawable = class sealed public
    const
      SemiBlack=$7f020001;
      violet=$7f020000;
    end;

    id = class sealed public
    const
```

```
      back=$7f070001;
      clear=$7f070003;
      editor=$7f070000;
      image=$7f070002;
    end;

    layout = class sealed public
    const
      skeleton_activity=$7f030000;
    end;

    strings = class sealed public
    const
      Back=$7f050001;
      Clear=$7f050002;
      Hello=$7f050003;
      skeletonapp=$7f050000;
    end;

    style = class sealed public
    const
      ActionButton=$7f060000;
      TextAppearance=$7f060001;
      TextAppearance_ActionButton=$7f060002;
    end;
  end;

implementation

end.
```

This translation mimics the structure of the Java original. But this is by no means necessary, since the prime purpose of this unit is to make the integer values assigned by the asset packer available. The following unit would do just as well:

```
unit Resources;

interface

Const
  ColorSemiBlack=$7f020001;
  Colorviolet=$7f020000;

  IDback=$7f070001;
  IDclear=$7f070003;
  IDeditor=$7f070000;
  IDimage=$7f070002;
  Layout_skeleton_activity=$7f030000;

  Strings_Back=$7f050001;
  Strings_Clear=$7f050002;
  Strings_Hello=$7f050003;
  Strings_skeletonapp=$7f050000;
```

```
  StyleActionButton=$7f060000;
  StyleTextAppearance=$7f060001;
  StyleTextAppearance_ActionButton=$7f060002;

implementation

end.
```

7 Pascal Code: the activity

The application manifest states that the activity to start when the skeleton application is started is `eu.blaisepascal.skeletonapp.TSkeletonActivity`. This is the name of the Activity class descendant that will be instantiated. The purpose and intent of this class is in fact equivalent to a form declaration in Delphi or Lazarus.

For the Skeleton Application, it is declared as follows:

```
TSkeletonActivity = class(AAActivity,AWButton.InnerOnClickListener)
protected
  FEditor : AWEditText;
  FClear : AWButton;
  FBack : AWButton;
  FBackID : Integer;
  FClearID : Integer;
public
  procedure onCreate(savedInstanceState: AOBundle); override;
  Function onCreateOptionsMenu(AMenu : AVMenu) :  boolean ; override;
  Function onPrepareOptionsMenu(AMenu : AVMenu) : Boolean; override;
  Function onOptionsItemSelected(AItem : AVMenuItem) : Boolean; override;
  Procedure onClick(v : AVView); // OnClick
end;
```

Similar to the purpose and intent, the declaration of this class is in many ways similar to a form declaration as created in Delphi and Lazarus.

8 Layout: form design

In Delphi and Lazarus, each form file has a resource file (a .dfm or .lfm) associated with it. The same can be done in Android: to define the layout and put controls (views, in Android speak) of an activity, a layout file is created in the resources. This resource file is an XML file, located with the other resource files, in the directory res/layout.

The format of an Android Layout file is not very complicated, but there are a lot of attributes that can be specified. In general, it mixes layouts with controls: each tag is either a control (a view) or a group of controls (a layout)

For the skeleton application, this layout file starts as follows:

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

```
  android:orientation="vertical">
  <EditText android:id="@+id/editor"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:autoText="true"
    android:capitalize="sentences"
    android:layout_weight="1"
    android:freezesText="true" >
    <requestFocus />
  </EditText>
```

The layout file starts out by specifying a linear layout. This means that the controls in this layout will be placed linearly, one next to the other. Several attributes control this layout (For clarity, the android XML namespace is skipped):

orientation this attribute has a value of 'vertical' meaning the controls will be placed from top to bottom.

layout_width and layout_height These attributes both have a value of `match_parent`, which corresponds to a `alClient` align property in Delphi: the activity will match the size of the parent view, which is the whole screen, since this is the main activity in the application.

Under the `LinearLayout` element, there is a `EditText` element. This is a text-editing element. It too has several attributes:

id A unique identifier for this control. The value of "@+id/editor" means that the resource packer will auto-generate an numerical id with name `editor`. Indeed, in the Java class for the skeleton application, we can see a value 'editor=$7f070000'.

autoText a value of true should autotext be enabled (completion).

capitalize a value of "sentences" means that the beginning of sentences will be automatically capitalized.

layout_weight a value between 0 and 1. The value 1 means that if the layouter needs to resize and re-layout controls, then this control will receive all extra space.

freezesText The value "true" tells the control to save its text.

Of these attributes, the 'id' attribute is the most important one: as will be seen, assigning an ID allows to fetch a reference to the run-time instance of this control.

Below the EditText control, a new – horizontal – LinearLayout is created, with 2 buttons and an image in between:

```
  <LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:gravity="center_horizontal"
    android:orientation="horizontal"
    android:background="@drawable/SemiBlack">
    <Button android:id="@+id/back"
      style="@style/ActionButton"
      android:text="@string/Back" />
```

```
    <ImageView android:id="@+id/image"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:paddingLeft="4dip"
      android_paddingRight="4dip"
      android:src="@drawable/violet" />
    <Button android:id="@+id/clear"
      style="@style/ActionButton"
      android:text="@string/Clear"
      android:textColor="@color/Red" />
  </LinearLayout>
</LinearLayout>
```

This time, the width and height of the layout are set to `wrap_content`, which corresponds roughly to `AutoSize:True` in a Delphi form. The `gravity` attribute tells the layouter that the layout is supposed to be centered horizontally in it's parent layout. By contrast, `layout_gravity` controls how the items are spread in the (inner) layout.

The background has a reference to a black color (defined in the color resources).

The buttons and image are positioned inside this layout: they all get a unique ID. Both buttons have a `style` as well as a `text` attribute: all these are references to resources: the styles are defined in a styles.xml resource file, the texts are in the strings.xml resource file. The text color of the `clear` button is also set through a resource.

The ImageView has an `src` attribute: this tells the renderer that the image is located in the `drawable` resources, with name `violet`.

9 Loading the layout

While similar in concept to the .dfm file in Delphi (or .lfm file in Lazarus) there are some major differences:

1. The layout of an activity is not automatically applied from the layout file.
2. controls specified in the layout do not automatically have a field associated with them in the activity class.
3. Last but not least: an activity layout does not contain event handlers.

All these automatically handled features of Delphi this must be done manually in an Android application. To do so, the `AAActivity` class has a `onCreate` method, which must be overridden and in which the necessary work must be done.

This is what the method looks like:

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

begin
  inherited;
  // Load layout
  setContentView(R.layout.skeleton_activity);
  // Load instances of controls.
  FEditor:=AWEditText(findViewById(R.id.editor));
  FBack:=AWButton(findViewById(R.id.back));
  FClear:=AWButton(findViewById(R.id.clear));
```

```
  // Set handlers and initialize:
  FBack.setOnClickListener(Self As AWButton.InnerOnClickListener);
  FClear.setOnClickListener(Self As AWButton.InnerOnClickListener);
  FEditor.setText(getText(R.strings.Hello));
end;
```

It starts by calling `SetContentView`, with the ID of the layout as generated by the asset packager. Then the generated IDs of the various controls are used to get references to the controls (buttons and editor).

Finally, an onClick handler is set on both buttons, and the text of the editor is initialized with a text from the resources: the `getText` function can be used to retrieve a text from the resources by ID.

The `AWButton.InnerOnClickListener` is an interface that must be used to listen to OnClick events for buttons. The `TSkeletonActivity` class specifies this interface as one of its interfaces. This particular interface specifies one method: `onClick` which is implemented as follows:

```
// OnClick interface
Procedure TSkeletonActivity.onClick(v : AVView);
begin
  if (V=Fback) then
    Finish
  else if (V=FClear) then
    FEditor.SetText(JLString(''));
end;
```

A simple method which - depending on the button clicked - exits the activity (using `Finish`) or clears the edit using `setText`.

Note that since an interface is used as an event handler, the name of the method is fixed. As a consequence only one method can be used for all buttons' onclick listeners: a class can implement an interface only once. This simply means that the `v` parameter must be used to determine which particular button called the handler.

Finally, the application creates 2 menu handlers which implement the same actions as the 2 buttons in the (standard) options meny of the activity.

The options menu is created in code by the Activity class, and the `OnCreateOptionsMenu` method is called to fill it. By overriding this method, 2 items are added to the menu:

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

begin
  inherited;
  AMenu.add(0,AVMenu.FIRST,0,R.strings.back).setShortcut('0','b');
  AMenu.add(0,AVMenu.FIRST+1,0,R.strings.clear).setShortcut('1','c');
  Result:=True;
end;
```

This code is quite self explaining, except maybe for the AVMenu.FIRST: Each option menu item has a number associated with it that uniquely identifies it. AVMenu.FIRST is a constant telling Android where the menu items should start counting:

As can be seen in the code, the text for the menu items is specified with a resource id: the same texts as used for the button captions is used. Additionally, each menu gets a shortcut assigned to it.

The 'Clear' menu item should be disabled when there is no text. This can be done by overriding the `OnPrepareOptionsMenu` method, which is the equivalent of the OnPopup menu handler in Delphi:

```
Function TSkeletonActivity.onPrepareOptionsMenu(AMenu : AVMenu) : Boolean;

begin
  Inherited;
  AMenu.findItem(AVMenu.FIRST+1).setVisible(FEditor.getText().length()>0);
  Result:=True;
end;
```

Again, the code is self explanatory: the menu item is searched through its identifier, and made invisible if there is no text in the editor.

The equivalent of the Delphi 'OnClick' menu item handler for an options menu is the `onOptionsItemSelected` method. It must be overridden to act on a click:

```
Function TSkeletonActivity.onOptionsItemSelected(AItem : AVMenuItem) : Boolean;

begin
  Case AItem.getItemId() of
  AVMenu.First:
    begin
    finish();
    Exit(true);
    end;
  AVMenu.First+1:
    begin
    FEditor.setText(JLString(''));
    Exit(true);
    end
  else
    Result:=Inherited onOptionsItemSelected(AItem);
  end;
end;
```

Note again the use of the unique ID of the menu item to determine which item was clicked.

10 Putting it all together

With this, the Object Pascal code for the skeleton application is finished. To create an actual Android application, we need to create an .apk file: a process that takes several steps. For people using Ant (a Java build system), there are ant scripts to create apk files - although Google has meanwhile changed to Gradle for it's build system.

The process of creating an .apk file can be done manually, and starts by compiling the pascal code to java bytecode with FPC:

```
ppcjvm -B -n -Tandroid -Fuusr/local/fpc/2.7.1/units/rtl-jvm \
  -FEbin/classes src/skeletonapp.pas
```

When all is compiled, the java bytecode must be converted to Dalvik executable code.

This is done with the 'dx' compiler included in the Android SDK:

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

The compiled Java bytecode is written to a file `bin/classes.dex`.

Two more steps remain: The first is to create an .apk file, this is done using apkbuilder:

```
apkbuilder bin/skeletonapp-unsigned.apk -u -z bin/skeletonapp.ap_ -f bin/classes.
```

The bin/skeletonapp.ap_ file is the resource file created by the aapt program (see the section on resources) The bin/classes.dex is the actual java bytecode, ready to run in Dalvik.

Note that the `apkbuilder` program (a batch file or shell script) is no longer offically supported, but it still works.

The last step is actually optional, but is needed if you want to place an application in google play: to sign your application using a public/private key mechanism.

```
jarsigner -keystore ~/.jarkeys -signedjar bin/skeletonapp.apk bin/skeletonapp-uns
```

The jarsigner is a java application that can be used to sign jar files (an .apk file is actually a jar file). For it to work, you need to create a public/private key pair.

When all this is done, the application can be uploaded to an Android virtual device (or connected device) with `adb` from the Android SDK:

```
adb -e install bin/skeletonapp.apk
```

If all went well, the apps screen of the Android device should look like figure 1 on page 15: When tapped, the application will look as in figure 2 on page 16:

11 Conclusion

This article attempts to show how to create a native Android application using Free Pascal to generate Java Bytecode. The process is initially not so simple, but once the environment is set up correctly, it becomes actually relatively easy.

Setting up the environment can be simplified by using makefiles (or some custom program that performs all the steps). The most difficult part is then mastering the Android idiom and making sure the layouts are all correct.

The Android API is huge. Not only are all standard Java classes available, the Android API offers many useful features: Not only visual classes, but many other classes are available as well: Classes for preference handling, text-to-speech conversion, handwriting recognition. One of these classes is an interface to the SQLite database: every Android device contains the SQLite engine which 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.

In a next article, the use of the Android database API and preferences API will be discussed.

Figure 1: The installed skeleton application

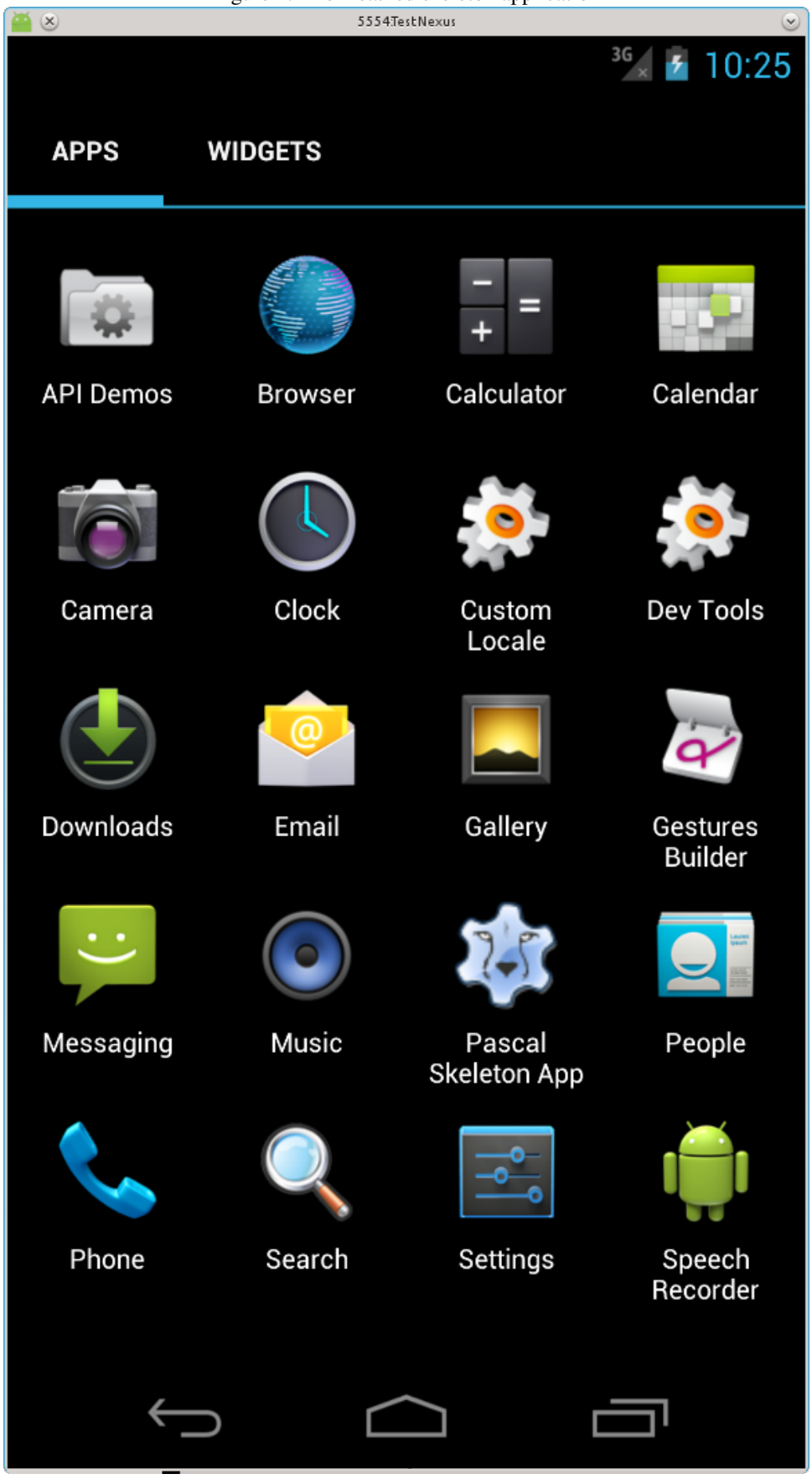

Figure 2: Running the skeleton application