Android: Run

Android is an open-source OS for smartphones, which makes it easy to write and publish software for it, as **Juliet Kemp** will now demonstrate.

The Nexus 1 phone that we reported on in this issue's news pages (see p6) is the only one of the current generation of smartphones that runs an an open-source operating system: Google's newly-developed Android. Android runs on a Linux kernel, with a bunch of Google-developed Java libraries sitting between that and the software. The open source development model means that it's straightforward to start developing and releasing your own software for the Nexus 1 or G1, written in Java and making use of the Android-specific libraries as well as standard Java libraries. Even if you don't have your own Android-running device, the dev tools are all available for free online, including a phone emulator to test your software on. (Although it is of course recommended that you test your software on a real device you release it into the wild.)

In this two-part series I'll take you through the steps involved in setting up your development environment, writing a very basic list-making application and then releasing it to the public. This month we'll look at the dev and test setup, and get the first version of the application working. So turn make a cup of tea, set your phone to silent mode and prepare to enter the not-at-all *Blade Runner*-esque world of Android development.

Our expert

Juliet Kemp fell madly in love with her G1 Android and now gets withdrawal symptoms when parted from it for longer than a few minutes.

Part 1 Setting up your dev environment

The first step is to download the Android SDK. There are basically two ways of going about your Android development: you can use *Eclipse* (the Java development platform) with an Android plugin, which will do a certain amount of work for you; or you can do it all yourself, which is still fairly straightforward. Here we'll tackle the non-*Eclipse* setup, partly to give you more control over exactly what's going on, and partly because *Eclipse* can be unusably slow on older machines (including my own desktop!).

Note that you'll need version 1.6 of Java to work with the most recent version of Android: this is available as the **openjdk-6-jdk** for Debian/Ubuntu. You may also need to run **sudo update-alternatives --config java** to set the correct version of Java to use.

The Android SDK is available from the Android developer site (**http://developer.android.com/index.html**): at the time of writing the current version was 1.5 rev 3. Download, unzip, and put it wherever you want to: for example, **/usr/local/android-sdk-linux_x86-1.5_r2/**. Check that the permissions are correct and that your user is in the correct group. After that, the only setup you need is to edit your **.bashrc** or **.bash_profile** to include the **tools/** directory in your **$PATH**:

```
export PATH=${PATH}:/usr/local/android-sdk-linux_x86-1.5_r2/tools/
```

Open a new terminal window (or type **source .bashrc**) to put this into effect. That's all you need to do: the environment is now ready to use. We'll set up the testbed and your emulated phone later on: first, let's generate an empty project so we've got somewhere to put our code once we start editing it. Create a working directory for your Android code and **cd** into it, then use the *android* tool to generate the new project:

```
mkdir ~/android/
cd ~/android
android create project --package com.example.list --activity List \\
 --target 2 --path ~/android/List
```

The **--package** option identifies the namespace for your new package. These follow the same rules as Java package namespaces. The basic rule is that you take your own internet domain name (or that of your organisation) and reverse this, component by component, to generate the package name. So if you have the **myname.com** domain, you can use **com.myname.list**. (If you don't have a domain, one option is to use **local.myname.list**, but this risks namespace collisions.)

The **--activity** argument sets the name of your main Activity class (I'll talk in more depth about Activities next month; for now, you just need to know that it's the name of the main class of your project). **--target** is the library that you're building against. **2** is the standard 1.5 library: use

with the robots

The Android emulator ready for action – there's a keyboard to the right of the pretend device, and it'll respond to mouse movements.

Developing with Eclipse

To use *Eclipse*, you'll still have to download the SDK, note where you put it, and set up your **$PATH** correctly.

You'll need at least version 3.3 of *Eclipse*, so you'll have to install from the *Eclipse* website). Once you have *Eclipse* up and running, go to Help > Install Software and type **https://dl-ssl.google.com/eclipse/android** into the Work With box (try **http://** if you have trouble with https://).

Select the checkbox next to Developer Tools under the Android plugin, then choose Install. On the next window, make sure that both the DDMS and the dev tools are due to be installed, then click Next, accept the licence agreement, and click Finish.

Android list targets to see what's available. **--path** is the project directory, which will be created for you if necessary.

Now type **cd ~/android/List** and have a quick look at the directory structure. Your source code is in **src/local/myname/list/**; the other important directory is **res/**, where package resources live. **AndroidManifest.xml** is the application manifest, which contains all the information about the application required by the Android system. It sets up the component structure of the application, declares appropriate provisions, declares the required libraries and specifies the minimum API that the software needs. (For more information on this file, check out the Developer docs.) The other useful file is **build.xml**, the build file for *Ant*, the build tool. You shouldn't need to touch either of these two files by hand.

So far there's no code in here that actually does anything. But the project setup generates class stubs for you, so we can compile it anyway. From the top directory, run:

```
ant debug
```

Check out **bin/**: there's now a file in there called **List-debug.apk**. You could install that on to the test emulator, except that we haven't set that up yet ,and anyway it won't do anything. The debug build target is used while you're developing; next month we'll look at what to do when you want to generate a real program.

Part 2 List.java

We're going to write a basic list-making application with a simple database back-end, where you can add an item to a list (and later delete it). First of all, let's set up the main class (**List.java**). This is what will be run when someone starts the app up, and at the moment all it will do is show a view of the existing list (so it needs to grab any existing items from the database), and create a menu option to add an item.

List extends **ListActivity** – this is an Android class that deals with displaying a list of items and managing various event handlers to react when the user selects or clicks on an item. An Activity is basically a class that deals with 'something that the user does' within your application. (So an application can have multiple Activities, although ours won't just yet!)

There are a couple of private variables and objects set up at the top of the class which we'll need later. The **onCreate** method is the method that runs when the class is created (ie when the app starts):

```
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
                mDb = new ListDbAdapter(this);
                mDb.open();
                getData();
                registerForContextMenu(getListView());
}
```

The **@Override** line is there because this overrides an inherited method. We'll look at the layout set with **R.layout**.main later. The rest of the method deals with setting up the interaction with the database, grabbing the data out of it, and registering for a context menu (the type of menu that shows up when you long-click on an item, which is the Android equivalent of right-clicking). **registerForContextMenu** is inherited from **ListActivity**.

The next thing is to write the **getData()** method, which will grab the data from the database:

```
    private void getData() {
                mCursor = mDb.fetchAllItems();
                startManagingCursor(mCursor);
                String[] cols = new String[]{
ListDbAdapter.DB_ITEM };
                int[] views   = new int[]{ R.id.text1 };
                SimpleCursorAdapter row_cursor =
```

Quick tip

If you're running on a slow computer you may find that the 1.5 version of the test environment is very slow. You can set up a 1.1 AVD using **-t 1** and use that for initial testing.

› Here's *DDMS* in action. Check out the grey!

```
                new
SimpleCursorAdapter(this, R.layout.list_row, mCursor, cols,
                                                views);
        setListAdapter(row_cursor);
    }
```

mCursor is one of our pre-established private objects: a Cursor allows access to objects returned by a database query (ie a database query returns a Cursor item – we'll look at the database methods later when we write our database interface class). **startManagingCursor** is another method inherited from **ListActivity**, which will let Android do the Cursor lifecycle management work for us.

The next line sets up a String array for our columns: in this case, we only have one, which is the **DB_ITEM** column from our database class (see below to write this!). The line after that sets up an **int** array to match the String array: each entry in the **int** array defines the View to which the corresponding column in the String array should be bound. Here, the **DB_ITEM** column is bound to the **R.id.text1** View. (Look for more about Views in the second part of this series. For now, the short version is that they're a way to manage an area of the screen.) The **SimpleCursorAdapter** sets up the layout for the rows (the **R.layout.list_row** layout), and links the **cols** String array and the views **int** array. **setListAdapter** ties this to the list view.

Next we create three menu methods. The first one adds items to the menu:

```
    public boolean onCreateOptionsMenu(Menu menu)
{
        super.onCreateOptionsMenu(menu);
        menu.add(NONE, ADD_ID, NONE,
R.string.menu_add);
        return true;
}
```

We only have one menu option: its label is defined with **R.string.menu_add** (we'll set that up in a moment!), and its positioning is defined by **ADD_ID** (which was set up top of the class to equal the **Menu.FIRST** constant, so it'll be the first option in the menu). The first **NONE** means that the item should not be in a group, and the second **NONE** means that we don't care about order.

Next we need to deal with what happens when the user clicks that menu item:

```
    public boolean onMenuItemSelected(int id,
MenuItem item) {
        switch(item.getItemId()) {
            case ADD_ID:
                createItem();
                return true;
        }
        return super.onMenuItemSelected(id,
item);
    }
```

The switch statement handles things depending on the ID of the menu option selected. Here we only have one, so it's pretty easy!

Finally, we need the method that creates the new item (ie, adds it to the database):

```
    private void createItem() {
        /*later we'll need to actually get a way of
getting data into this!*/
        mDb.createItem(getString(R.string.new_
item));
        getData();
    }
```

We're going to leave the more complicated bit of handling user input until the second article in this series: for now, all we do is add an entry to the database that contains the text stored in the **R.string.new_item** resource (see below). Then we grab all the data from the database again to redraw the screen, so that the new item will appear as soon as it's added.

Now let's take a look at those R strings and layouts that we've made references to.

Strings and things

If you take a look at your **List/** dev directory, there's a subdirectory there called **res/**. This is where you keep all your resources. Resources are what Android calls pretty much anything external to the code that you want to reference: images, layouts, string data and that sort of thing.

In this project so far we only have a layout and a couple of strings to worry about. The layout will be kept in **res/layout/**, and the string data is in **res/values/strings.xml**. You want to edit this file so it looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">List</string>
    <string name="menu_add">Add item</string>
    <string name="new_item">New item</string>
</resources>
```

As you can see, it's an XML file with a pretty straightforward format. We've set the application name and two strings that are used in the **List.java** file. These are referred to as **R.string.menu_add** and **R.string.new_item**, and you've already seen all of them used in the code above. It's good practice to put all your constant strings in this file: it's more efficient, it makes life simpler if you ever want to change anything, and when in due course Android supports internationalisation and localisation, you'll be in a better position to set your app up to use that.

The other sort of resource we've used so far is a layout resource. The main layout is in **res/layout/main.xml**, which should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/
apk/res/android"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
        <ListView android:id="@+id/android:list"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
<TextView android:id="@+id/android.empty"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="No items in this list"
    />
</LinearLayout>
```

Quick tip

Resources are returned as a **CharSequence**: if you need to be sure that you're getting a string, use the **getString()** method as shown in **createItem()**. You may not always need this, but it's useful if you get a resource-related compile error.

You may notice that we've got a fixed string in there. Better practice would be to replace that line with this one:

```
android:text="@string/empty_list"
```

The **@** identifies this as a reference, and because it's in this package you don't need to specify anything other than that it's a string, and the name of the string. Now add an appropriate line to **res/values/strings.xml**:

```
<string name="empty_list">No items in this list</string>
```

and you've correctly externalised your string! The other file we've referred to in our code is **res/layout/list_row.xml**:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/text1" xmlns:android="http://
schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
/>
```

That **text1** field is used in the **getData** method:

```
int[] fields  = new int[]{ R.id.text1 };
```

where it was used to link that view (**R.id.text1**, which here is set up as a **TextView** with wrapping widthwise and heightwise) to a particular column. So what we've done is said: for that column, we use this view, but we've defined the view in the resources section rather than in the code. This is a bit like putting your HTML visual setup in CSS: it makes it easier to modify the layout if you choose, and it makes it easier to give your app a visual identity.

Quick tip

The docs claim that you need to run **logcat** from *DDMS* to see the logs, but in fact this just gives you a second log window (which doesn't have the useful colours in the one on the main screen) showing the same info.

Part 3 Setting up your database & testing

The code for the database is in **ListDbAdapter**. The start of this file sets up various constants and private objects, including a string that will create a database with two fields: an integer key field and a text item field:

```
private static final String DB_CREATE =
  "create table list (_id integer primary key autoincrement, "
    + "item text not null);";
```

We also set up a private **DatabaseHelper** class, which inherits from the **android.database.sqlite.SQLiteOpenHelper** class. This will do the actual interacting with the database for us, and uses that **DB_CREATE** string. There's also a method in there to deal with upgrades.

When creating the **ListDbAdapter**, all we need is the Context: this is a class provided by the Android system that acts as an interface to information about the application environment. We also set up **open** and **close** methods, which do what you'd expect.

The **createItem** method is the next interesting one:

```
public long createItem(String item) {
    ContentValues content = new ContentValues();
    content.put(DB_ITEM, item);
   return mDb.insert(DB_TABLE, null, content);
}
```

ContentValues is effectively a hash implementation: the item value is stored with **DB_ITEM** as a key and then the **DatabaseHelper mDb** uses the hash when interacting with the database and add the given content. There's a delete method as well, and a method to return all items (**fetchAllItems**).

OK, so now let's set up our test environment. The first thing to do is to generate an Android Virtual Device (AVD): this is the pretend phone that the emulator will run. You can generate multiple AVDs, so you can create different phone setups and have their data independently maintained. Anything you save to an AVD when the emulator is running will survive between emulator runs. For a basic test phone running the current version of Android, use:

```
android create avd -n my_avd_1.5 -t 2
```

The targets are the same as when you were generating the project: start off testing against the same target that you built the project for, although later you may wish to generate different AVDs to different targets to see if your software is backwards and/or forwards compatible. You're asked if you want to set any hardware options; say **no** to get the defaults. The AVD will be saved to **~/.android/avd**.

Next, start up the emulator:

```
emulator -avd my_avd_1.5
```

Finally, install your software on the device:

```
adb install ~/android/List/bin/List-debug.apk
```

Once that command has run, click on the tab at the bottom of the phone screen and scroll down the menu that appears until you find the List item. Click on that and it should run.

If you make a change to the code, recompile it and want to reinstall it, you need to use the **-r** switch to *android* install:

```
adb install -r ~/android/List/bin/List-debug.apk
```

The app will remain installed if you shut the emulator down and start it up again with the same AVD: all the existing information is stored as part of the AVD on shutdown.

Debugging

On this occasion your code should run perfectly, but sadly that isn't always the case. The Android emulator won't give you much information: you need to run the debugger, *DDMS*. This will automatically connect itself to the running emulator, and there's a logging window at the bottom of the screen that you can use to check stack traces and exceptions. To output something to the log from your app, use this syntax:

```
import android.util.Log;
private static final String TAG = "List";
Log.i(TAG, "List.getData() - about to talk to database");
Log.w(TAG, "List.getData() - oh dear, something has gone
wrong");
```

Use the **TAG** string as a label for which activity you're logging from (here, the **List** activity; you could also log from the **ListDbAdapter** class and use that as a tag). Use **Log.d** for debug, **Log.i** for information, **Log.e** for error and **Log.w** for warning. LXF

Resources

The main resource is the developers' guide and documentation that comes with the Android SDK. There's a lot of useful information in here, and some sample code: I found it incredibly helpful when getting started with Android coding.

There are also a couple of lists on Google Groups (android-beginners, android-developers, and android-discuss): check the description of each before you post to make sure you're in the appropriate forum for your question.

If you use IRC, check out the #android channel on the **irc.freenode.net** server. There are also various Android forums which each have their own developer boards.

» **Next month** We'll examine more details of the Android application model.