Notepad Exercise 1

In this exercise, you will construct a simple notes list that lets the user add new notes but not edit them. The exercise demonstrates:

- *The basics of `ListActivities` and creating and handling menu options.*
- *How to use a SQLite database to store the notes.*
- *How to bind data from a database cursor into a ListView using a SimpleCursorAdapter.*
- *The basics of screen layouts, including how to lay out a list view, how you can add items to the activity menu, and how the activity handles those menu selections.*

[Exercise 1] [Exercise 2] [Exercise 3] [Extra Credit]

Step 1

Open up the `Notepadv1` project in Eclipse.

`Notepadv1` is a project that is provided as a starting point. It takes care of some of the boilerplate work that you have already seen if you followed the Hello, World tutorial.

1. Start a new Android Project by clicking **File** > **New** > **Android Project**.
2. In the New Android Project dialog, select **Create project from existing source**.
3. Click **Browse** and navigate to where you copied the `NotepadCodeLab` (downloaded during setup) and select `Notepadv1`.
4. The Project Name and other properties should be automatically filled for you. You must select the Build Target—we recommend selecting a target with the lowest platform version available. Also add an integer to the Min SDK Version field that matches the API Level of the selected Build Target.
5. Click **Finish**. The `Notepadv1` project should open and be visible in your Eclipse package explorer.

If you see an error about `AndroidManifest.xml`, or some problems related to an Android zip file, right click on the project and select **Android Tools** > **Fix Project Properties**. (The project is looking in the wrong location for the library file, this will fix it for you.)

Step 2

Take a look at the `NotesDbAdapter` class — this class is provided to encapsulate data access to a SQLite database that will hold our notes data and allow us to update it.

At the top of the class are some constant definitions that will be used in the application to look up data from the proper field names in the database. There is also a database creation string defined, which is used to create a new database schema if one doesn't exist already.

Accessing and modifying data

For this exercise, we are using a SQLite database to store our data. This is useful if only *your* application will need to access or modify the data. If you wish for other activities to access or modify the data, you have to expose the data using a `ContentProvider`.

If you are interested, you can find out more about content providers or the whole subject of Data

Storage. The NotePad sample in the `samples/` folder of the SDK also has an example of how to create a ContentProvider.

Our database will have the name `data`, and have a single table called `notes`, which in turn has three fields: `_id`, `title` and `body`. The `_id` is named with an underscore convention used in a number of places inside the Android SDK and helps keep a track of state. The `_id` usually has to be specified when querying or updating the database (in the column projections and so on). The other two fields are simple text fields that will store data.

The constructor for `NotesDbAdapter` takes a Context, which allows it to communicate with aspects of the Android operating system. This is quite common for classes that need to touch the Android system in some way. The Activity class implements the Context class, so usually you will just pass `this` from your Activity, when needing a Context.

The `open()` method calls up an instance of DatabaseHelper, which is our local implementation of the SQLiteOpenHelper class. It calls `getWritableDatabase()`, which handles creating/opening a database for us.

`close()` just closes the database, releasing resources related to the connection.

`createNote()` takes strings for the title and body of a new note, then creates that note in the database. Assuming the new note is created successfully, the method also returns the row `_id` value for the newly created note.

`deleteNote()` takes a *rowId* for a particular note, and deletes that note from the database.

`fetchAllNotes()` issues a query to return a `Cursor` over all notes in the database. The `query()` call is worth examination and understanding. The first field is the name of the database table to query (in this case `DATABASE_TABLE` is "notes"). The next is the list of columns we want returned, in this case we want the `_id`, `title` and `body` columns so these are specified in the String array. The remaining fields are, in order: `selection`, `selectionArgs`, `groupBy`, `having` and `orderBy`. Having these all `null` means we want all data, need no grouping, and will take the default order. See `SQLiteDatabase` for more details.

> **Note:** A Cursor is returned rather than a collection of rows. This allows Android to use resources efficiently -- instead of putting lots of data straight into memory the cursor will retrieve and release data as it is needed, which is much more efficient for tables with lots of rows.

`fetchNote()` is similar to `fetchAllNotes()` but just gets one note with the *rowId* we specify. It uses a slightly different version of the `SQLiteDatabase` `query()` method. The first parameter (set *true*) indicates that we are interested in one distinct result. The *selection* parameter (the fourth parameter) has been specified to search only for the row "where _id =" the *rowId* we passed in. So we are returned a Cursor on the one row.

And finally, `updateNote()` takes a *rowId*, *title* and *body*, and uses a `ContentValues` instance to update the note of the given *rowId*.

Step 3

Open the `notepad_list.xml` file in `res/layout` and take a look at it. (You may have to hit the *xml* tab, at the bottom, in order to view the XML markup.)

This is a mostly-empty layout definition file. Here are some things you should know about a layout file:

- All Android layout files must start with the XML header line: `<?xml version="1.0" encoding="utf-8"?>`.
- The next definition will often (but not always) be a layout definition of some kind, in this case a `LinearLayout`.
- The XML namespace of Android should always be defined in the top level component or layout in the XML so that

Layouts and activities

Most Activity classes will have a layout associated with them. The layout will be the "face" of the Activity to the user. In this case our layout will take over the whole screen and provide a list of notes.

Full screen layouts are not the only option for an Activity however. You might also want to use a floating layout (for example, a dialog or alert), or perhaps you don't need a layout at all (the Activity will be invisible to the user unless you specify some kind of layout for it to use).

`android:` tags can be used through the rest of the file:

`xmlns:android="http://schemas.android.com/apk/res/android"`

Step 4

We need to create the layout to hold our list. Add code inside of the `LinearLayout` element so the whole file looks 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="@android:id/list"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
  <TextView android:id="@android:id/empty"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/no_notes"/>

</LinearLayout>
```

- The @ symbol in the id strings of the `ListView` and `TextView` tags means that the XML parser should parse and expand the rest of the id string and use an ID resource.
- The `ListView` and `TextView` can be thought as two alternative views, only one of which will be displayed at once. ListView will be used when there are notes to be shown, while the TextView (which has a default value of "No Notes Yet!" defined as a string resource in `res/values/strings.xml`) will be displayed if there aren't any notes to display.
- The `list` and `empty` IDs are provided for us by the Android platform, so, we must prefix the `id` with `android:` (e.g., `@android:id/list`).
- The View with the `empty` id is used automatically when the `ListAdapter` has no data for the ListView. The ListAdapter knows to look for this name by default. Alternatively, you could change the default empty view by using `setEmptyView(View)` on the ListView.

  More broadly, the `android.R` class is a set of predefined resources provided for you by the platform, while your project's `R` class is the set of resources your project has defined. Resources found in the `android.R` resource class can be used in the XML files by using the `android:` name space prefix (as we see here).

Step 5

To make the list of notes in the ListView, we also need to define a View for each row:

1. Create a new file under `res/layout` called `notes_row.xml`.
2. Add the following contents (note: again the XML header is used, and the first node defines the Android XML namespace)

Resources and the R class

The folders under res/ in the Eclipse project are for resources. There is a specific structure to the folders and files under res/.

Resources defined in these folders and files will have corresponding entries in the R class allowing them to be easily accessed and used from your application. The R class is automatically generated using the contents of the res/ folder by the eclipse plugin (or by aapt if you use the command line tools). Furthermore,

```
<?xml version="1.0" encoding="utf-8"?>
<TextView android:id="@+id/text1"

xmlns:android="http://schemas.android.co
m/apk/res/android"
    android:layout_width="wrap_content"

android:layout_height="wrap_content"/>
```

they will be bundled and deployed for you as part of the application.

This is the View that will be used for each notes title row — it has only one text field in it.

In this case we create a new id called `text1`. The **+** after the **@** in the id string indicates that the id should be automatically created as a resource if it does not already exist, so we are defining `text1` on the fly and then using it.

3. Save the file.

Open the `R.java` class in the project and look at it, you should see new definitions for `notes_row` and `text1` (our new definitions) meaning we can now gain access to these from the our code.

Step 6

Next, open the `Notepadv1` class in the source. In the following steps, we are going to alter this class to become a list adapter and display our notes, and also allow us to add new notes.

`Notepadv1` will inherit from a subclass of `Activity` called a `ListActivity`, which has extra functionality to accommodate the kinds of things you might want to do with a list, for example: displaying an arbitrary number of list items in rows on the screen, moving through the list items, and allowing them to be selected.

Take a look through the existing code in `Notepadv1` class. There is a currently an unused private field called `mNoteNumber` that we will use to create numbered note titles.

There are also three override methods defined: `onCreate`, `onCreateOptionsMenu` and `onOptionsItemSelected`; we need to fill these out:

- `onCreate()` is called when the activity is started — it is a little like the "main" method for an Activity. We use this to set up resources and state for the activity when it is running.
- `onCreateOptionsMenu()` is used to populate the menu for the Activity. This is shown when the user hits the menu button, and has a list of options they can select (like "Create Note").
- `onOptionsItemSelected()` is the other half of the menu equation, it is used to handle events generated from the menu (e.g., when the user selects the "Create Note" item).

Step 7

Change the inheritance of `Notepadv1` from `Activity` to `ListActivity`:

```
public class Notepadv1 extends ListActivity
```

Note: you will have to import `ListActivity` into the Notepadv1 class using Eclipse, **ctrl-shift-O** on Windows or Linux, or **cmd-shift-O** on the Mac (organize imports) will do this for you after you've written the above change.

Step 8

Fill out the body of the `onCreate()` method.

Here we will set the title for the Activity (shown at the top of the screen), use the `notepad_list` layout we created in XML, set up the `NotesDbAdapter` instance that will access notes data, and populate the list with the available note titles:

1. In the `onCreate` method, call `super.onCreate()` with the `savedInstanceState` parameter that's passed in.
2. Call `setContentView()` and pass `R.layout.notepad_list`.
3. At the top of the class, create a new private class field called `mDbHelper` of class `NotesDbAdapter`.
4. Back in the `onCreate` method, construct a new `NotesDbAdapter` instance and assign it to the `mDbHelper` field (pass `this` into the constructor for `DBHelper`)
5. Call the `open()` method on `mDbHelper` to open (or create) the database.
6. Finally, call a new method `fillData()`, which will get the data and populate the ListView using the helper — we haven't defined this method yet.

`onCreate()` should now look like this:

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.notepad_list);
        mDbHelper = new NotesDbAdapter(this);
        mDbHelper.open();
        fillData();
    }
```

And be sure you have the `mDbHelper` field definition (right under the mNoteNumber definition):

```
    private NotesDbAdapter mDbHelper;
```

Step 9

Fill out the body of the `onCreateOptionsMenu()` method.

We will now create the "Add Item" button that can be accessed by pressing the menu button on the device. We'll specify that it occupy the first position in the menu.

1. In `strings.xml` resource (under `res/values`), add a new string named "menu_insert" with its value set to `Add Item`:

   ```
   <string name="menu_insert">Add
   Item</string>
   ```

   Then save the file and return to `Notepadv1`.
2. Create a menu position constant at the top of the class:

   ```
   public static final int INSERT_ID = Menu.FIRST;
   ```

More about menus

The notepad application we are constructing only scratches the surface with menus.

You can also add shortcut keys for menu items, create submenus and even add menu items to other applications!.

3. In the `onCreateOptionsMenu()` method, change the `super` call so we capture the boolean return as `result`. We'll return this value at the end.
4. Then add the menu item with `menu.add()`.

The whole method should now look like this:

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        boolean result = super.onCreateOptionsMenu(menu);
        menu.add(0, INSERT_ID, 0, R.string.menu_insert);
        return result;
    }
```

The arguments passed to `add()` indicate: a group identifier for this menu (none, in this case), a unique ID (defined above), the order of the item (zero indicates no preference), and the resource of the string to use for the item.

Step 10

Fill out the body of the `onOptionsItemSelected()` method:

This is going to handle our new "Add Note" menu item. When this is selected, the `onOptionsItemSelected()` method will be called with the `item.getId()` set to `INSERT_ID` (the constant we used to identify the menu item). We can detect this, and take the appropriate actions:

1. The `super.onOptionsItemSelected(item)` method call goes at the end of this method — we want to catch our events first!
2. Write a switch statement on `item.getItemId()`.

   In the case of *INSERT_ID*, call a new method, `createNote()`, and return true, because we have handled this event and do not want to propagate it through the system.
3. Return the result of the superclass' `onOptionsItemSelected()` method at the end.

The whole `onOptionsItemSelect()` method should now look like this:

```
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        case INSERT_ID:
            createNote();
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
```

Step 11

Add a new `createNote()` method:

In this first version of our application, `createNote()` is not going to be very useful. We will simply create a new note with a title assigned to it based on a counter ("Note 1", "Note 2"...) and with an empty body. At present we have no way of editing the contents of a note, so for now we will have to be content making one with some default values:

1. Construct the name using "Note" and the counter we defined in the class: `String noteName = "Note " + mNoteNumber++`
2. Call `mDbHelper.createNote()` using `noteName` as the title and `""` for the body
3. Call `fillData()` to populate the list of notes (inefficient but simple) — we'll create this method next.

The whole `createNote()` method should look like this:

```
    private void createNote() {
        String noteName = "Note " + mNoteNumber++;
        mDbHelper.createNote(noteName, "");
        fillData();
    }
```

Step 12

Define the `fillData()` method:

This method uses `SimpleCursorAdapter,` which takes a database `Cursor` and binds it to fields provided in the layout. These fields define the row elements of our list (in this case we use the `text1` field in our `notes_row.xml` layout), so this allows us to easily populate the list with entries from our database.

List adapters

Our example uses a `SimpleCursorAdapter` to bind a database `Cursor` into a ListView, and this is a common way to use a `ListAdapter`. Other options exist like `ArrayAdapter` which can be used to take a List or Array of in-memory data and bind it in to a list as well.

To do this we have to provide a mapping from the `title` field in the returned Cursor, to our `text1` TextView, which is done by defining two arrays: the first a string array with the list of columns to map *from* (just "title" in this case, from the constant `NotesDbAdapter.KEY_TITLE`) and, the second, an int array containing references to the views that we'll bind the data *into* (the `R.id.text1` TextView).

This is a bigger chunk of code, so let's first take a look at it:

```
    private void fillData() {
        // Get all of the notes from the database and create the item list
        Cursor c = mDbHelper.fetchAllNotes();
        startManagingCursor(c);

        String[] from = new String[] { NotesDbAdapter.KEY_TITLE };
        int[] to = new int[] { R.id.text1 };

        // Now create an array adapter and set it to display using our row
        SimpleCursorAdapter notes =
            new SimpleCursorAdapter(this, R.layout.notes_row, c, from, to);
        setListAdapter(notes);
    }
```

Here's what we've done:

1. After obtaining the Cursor from `mDbHelper.fetchAllNotes()`, we use an Activity method called `startManagingCursor()` that allows Android to take care of the Cursor lifecycle instead of us needing to worry about it. (We will cover the implications of the lifecycle in exercise 3, but for now just know that this allows Android to do some of our resource management work for us.)

2. Then we create a string array in which we declare the column(s) we want (just the title, in this case), and an int array that defines the View(s) to which we'd like to bind the columns (these should be in order, respective to the string array, but here we only have one for each).
3. Next is the SimpleCursorAdapter instantiation. Like many classes in Android, the SimpleCursorAdapter needs a Context in order to do its work, so we pass in `this` for the context (since subclasses of Activity implement Context). We pass the `notes_row` View we created as the receptacle for the data, the Cursor we just created, and then our arrays.

In the future, remember that the mapping between the **from** columns and **to** resources is done using the respective ordering of the two arrays. If we had more columns we wanted to bind, and more Views to bind them in to, we would specify them in order, for example we might use `{ NotesDbAdapter.KEY_TITLE, NotesDbAdapter.KEY_BODY }` and `{ R.id.text1, R.id.text2 }` to bind two fields into the row (and we would also need to define text2 in the notes_row.xml, for the body text). This is how you can bind multiple fields into a single row (and get a custom row layout as well).

If you get compiler errors about classes not being found, ctrl-shift-O or (cmd-shift-O on the mac) to organize imports.

Step 13

Run it!

1. Right click on the `Notepadv1` project.
2. From the popup menu, select **Run As** > **Android Application**.
3. If you see a dialog come up, select Android Launcher as the way of running the application (you can also use the link near the top of the dialog to set this as your default for the workspace; this is recommended as it will stop the plugin from asking you this every time).
4. Add new notes by hitting the menu button and selecting *Add Item* from the menu.

Solution and Next Steps

You can see the solution to this class in `Notepadv1Solution` from the zip file to compare with your own.

Once you are ready, move on to Tutorial Exercise 2 to add the ability to create, edit and delete notes.

← Back to Notepad Tutorial   ↑ Go to top

Except as noted, this content is licensed under Creative Commons Attribution 2.5. For details and restrictions, see the Content License.
Privacy & Terms - Brand Guidelines - Report Document Issues