PART IV NOTEPAD TUTORIAL

Table of Contents

Goal....2
Scenario....2
Technology and Concepts....3
  Content Provider (CP)....3
  Uniform Resource Identifier....6
First Example....7
Binding....10
Notepad Tutorial....12
  The Tutorial – Preliminaries....12
    Class Note....12
    Class (ListActivity) ListNote....13
  Class NotesProvider....17
  Exercise 1....20
    Class NotesProvider - Open tasks....21
    Class NoteEdit – Open tasks....21
  Exercise 2....21
  Exercise 3....21

Illustration Index

Illustration 1: Individual Data Access....4
Illustration 2: Android Content Provider....4
Illustration 3: Objective of a standardised Data Access Layer....5
Illustration 4: Example....5
Illustration 5: URI - example....6
Illustration 6: CP and Binding....10

Goal

This more comprehensive and advanced exercise deepens a couple of important concepts as they can be found in the Android platform, namely the Activity, Intent and View concept. After finishing Part III, you should be already a bit more familiar with these concepts.

Summarising, Activities are components and represent the application concept. Each activity has a well defined lifecyle and is managed by a process in the underlying operation system. Activitities are put on a stack that allows, for example, to navigate backwards through Activities and pass data. Intents are messages and passed between Activties, they are either implicit or explicit, whereas the implicit Intents require an Intent-filter. A View is the main UI element and are associated to an Activity.

The goal is to implement a simple application to manage notices. As a possible extension it would be nice to associate multimedia content like Images, Audio, and Video (IAV) with a note.
The Android documentation also provides a Notepad Tutorial[1] and to ease learning, this exercise is oriented on the Android Notepad Tutorial. Of course we slightly modified some things. You will either have to extend this application or fill out a couple of gaps where implementation is missing.

You will also find the Android project Notepad-Students in your workspace, if not imported yet, do it as described in Part II!

The scenario is explained in the next section. A Technology and Concepts section follows to summarise and explain the Content Provider and Uniform Resource Identifier (URI) concept. Then you find a wee example. Binding is explained in section Binding and the last section is the Tutorial itself. The Tutorial explains the main parts of the Notepad project. At the end, your challenge is to complete the Notepad project as not all the functionality as required has been implemented yet. Bear in mind, the Notepad project may throw exceptions at its current state. We strongly recommend to read this complete handout first, at least skim it, and start afterwards. You find the exercise at the end of this document.

Scenario

Students using Android should be able to make notes for some subject. Beside the **date**, the **lecture**, a **lecturer**, a **title**, and the **actual note** should be stored. In an extended version, it would be nice to even link Image, Audio, and Video (IAV) content with a note. By contrast to the Android Notepad Tutorial this tutorial requires **master data**, namely the **lectures** and the **associated lecturer**. In total, your application requires **three tables: note, lecture, and lecturer.**

[1] http://developer.android.com/resources/tutorials/notepad/index.html (2010-11-13)

Technology and Concepts

Although already introduced in the presentation, the Content Provider (CP) concept is briefly introduced here again. In the last section, we have discussed the parts that are already implemented in the sample application. This should provide you enough background information to successfully finish this task. And, please use the Android documentation, specifically the Notepad Tutorial[2].

Content Provider (CP)

Data Access frameworks like Object Oriented Mapping (ORM) frameworks are often used with the goal to ease data access and decouple data access and persistence issues from the actual application logic.

If such frameworks are missing developers will have to implement their Data Access (DA) in their own way. We do not aim a discussion about DA technologies and their added value to software development and the quality of software, it is just something developers have to cope with, or should be aware of, and there are indeed a couple of advantages to use a standardised way to access data. Beside complex transformations, for example, from a relational model into an object oriented, as it can be found in ORM frameworks, such DA frameworks allow to decouple the DA from the actual application logic and thereby allowing for a separation of concerns.

By following such an approach an Facebook application, for example, can access the local Contacts via a Contacts CP to synchronise contacts, for instance. Generally, the goal is to enable a cross application data sharing without requiring the actual application to be part of the data access. So, via a well defined interface **application N** can access the data of **application M** and **O**. Of course, there are security issues and therefore an application requires the permission to access the data, but beside this “solvable”[3] issue, such an approach significantly allows to easier compose existing data.

One recurring pattern, despite the platform used, is to access the layer via an interface that provides the essential CRUD (Create, Read, Update, and Delete) operations for a specific Data Object (DO).

Conceptionally, Android follows a similar way and the interface (IF) CP defines the methods:

- `query(Uri, String[], String, String[], String)` which returns data to the caller
- `insert(Uri, ContentValues)` which inserts new data into the content provider
- `update(Uri, ContentValues, String, String[])` which updates existing data in the content provider
- `delete(Uri, String, String[])` which deletes data from the content provider
- `getType(Uri)` which returns the MIME type of data in the content provider

Hence, each CP has to implement these methods. Additionally, beside an interface the following must be defined or managed by the framework:

[2] http://developer.android.com/resources/tutorials/notepad/index.html (2010-11-12)
[3] Although security issues are always difficult to handle, especially in large scale heterogeneous applications, they are manageable in homogeneous and small application platforms, like the Android platform. Of course, this is a more fuzzy statement and even, or especially, mobile applications are often subject of security and privacy discussions, but in this context of data access, we think security or privacy is a manageable concern.

- Name of the CP
- Which CP provides which data type
- Which operations are made public
- Access control
- Concurrency

Data Access – Opportunity I

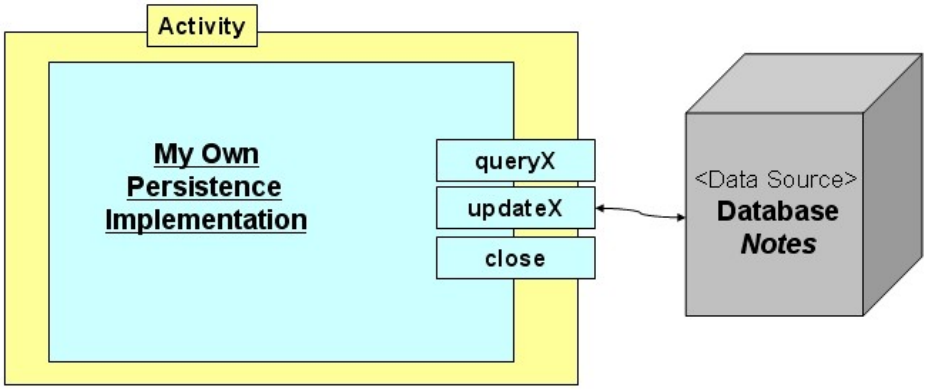

Illustration 1: Individual Data Access

Data Access – Opportunity II

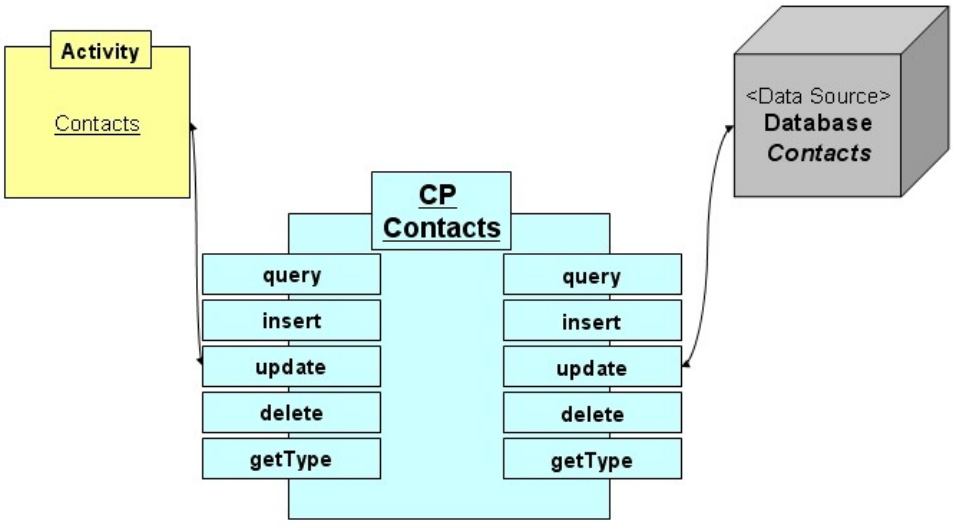

Illustration 2: Android Content Provider

Data Access - Objective

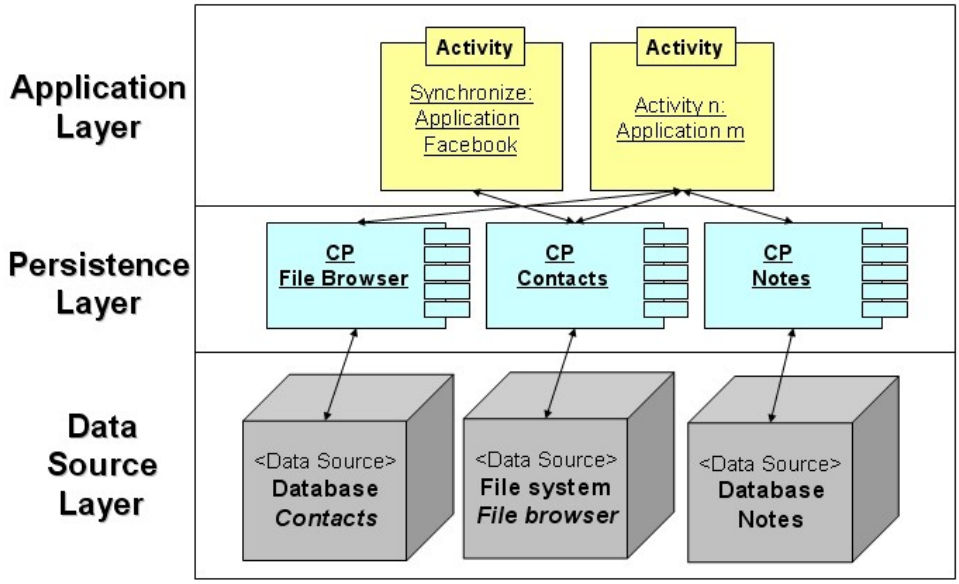

Illustration 3: Objective of a standardised Data Access Layer

Content Provider (CP) - I

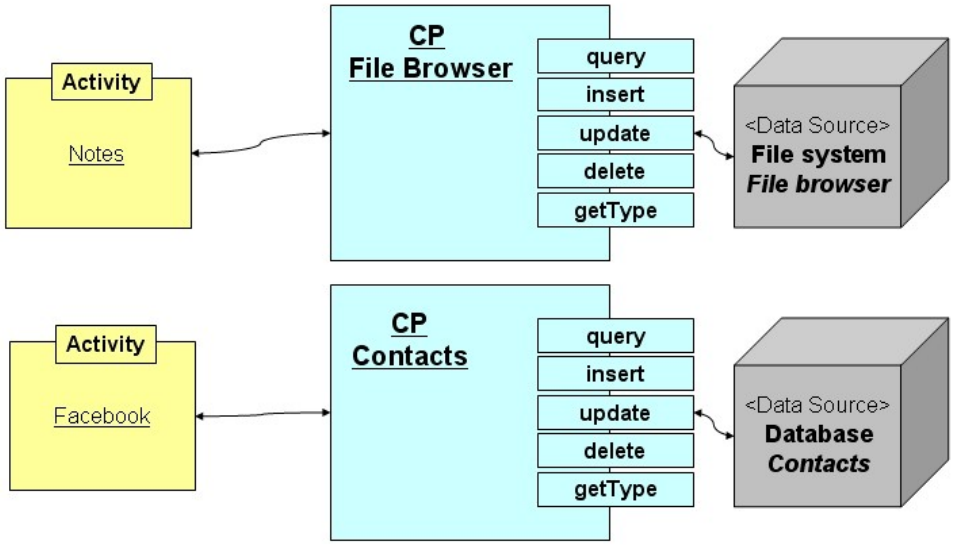

Illustration 4: Example

Uniform Resource Identifier

RFC 2396[4] defines an URI as: *"A Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource.[…]"*
Thus, an URI defines physical and abstract resources. Whereas a file is something physical, a service is something abstract. In Android an URI can define a CP, and also what data (table) needs to be queried or manipulated. For example, see illustration 5[5]:

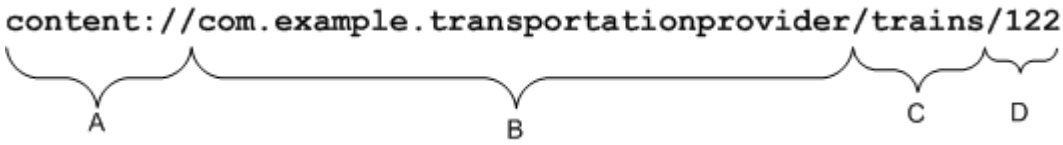

Illustration 5: URI - example

An URI referring to some content (CP) has to start with `content://`(part A). The next part of an URI, part B, directly refers to the implementation which is the full class name (package name and class name) of the provider. Part C is called **authority part** and in this example the CP allows to access information about the entity train. It is a good pattern to name the authority meaningful. The authority must be defined in the AndroidManifest.xml, too.

```
<provider android:name="NotesProvider"
android:authorities="de.rt.notes.notesprovider"/>
```

Please notice further, a CP can provide access to different entities, i.e., tables and so the same CP can provide access to notes and lectures or trains and cars. Even a hierarchical structure is feasible, e.g., `/transportation/land/trains` or `/transportation/sea/ships`.

The last part of an URI refers always to the unique identifier of a resource, e.g. the train with id 122.

NOTICE:
Per specification Android expects the name ID_ for the Primary-Key columns. ID_ is always defined in the interface `BaseColumns`[6]

The Android documentation suggests to define a constant for an URI, for example, the class Contacts defines a constant `CONTENT_URI` for the phone number and another one for the photos of the contacts.

```
android.provider.Contacts.Phones.CONTENT_URI
android.provider.Contacts.Photos.CONTENT_URI
```

Later on, in the last section, you will find a CONTENT_URI in `NotesProvider`:

[4]See http://www.faqs.org/rfcs/rfc2396.html
[5]See http://developer.android.com/images/content_uri.png
[6] See http://developer.android.com/reference/android/provider/BaseColumns.html

```
public static final Uri CONTENT_URI =
Uri.parse("content://de.rt.notes.notesprovider/notes");
```

To resolve an URI, which also means to retrieve the CP, the `ContentResolver` is necessary and Activity implements the method `getContentResolver(...)`.
About a `ContentResolver` the documentation states:[7] *"The URI constant is used in all interactions with the content provider. Every ContentResolver method takes the URI as its first argument. It's what identifies which provider the ContentResolver should talk to and which table of the provider is being targeted."*

First Example

This first example shows you how to use the Contact CP to open and show some details of a contact. You will find the project **openX** in your workspace, if not, import it. The complete code snippet can be found at the end of this section, too.

First of all, an Activity needs to be created with a corresponding layout. The layout has two `TextView` elements, one to show the name, the other to show the phone number of the contact. The following code lines show how to set the `contentView` and the `TextView` elememts.

```
setContentView(R.layout.openx);
TextView tv_name = (TextView) findViewById(R.id.tv_name);
TextView tv_mail = (TextView) findViewById(R.id.tv_mail);
```

To access the CP the URI of the contacts CP is required.

```
Uri contacts =  People.CONTENT_URI;
```

Now, to get a contact the CP needs to be queried and a cursor is returned. The method `managedQuery(...)` of the class Activity returns the cursor and in turn calls the method `query()` as implemented in the corresponding CP.

```
Cursor cur= managedQuery(contacts, null, null, null, People.NAME +" ASC");
```

The method managedQuery is defined as follows:

```
public final Cursor managedQuery  (Uri uri, String[] projection, String
        selection, String[] selectionArgs, String sortOrder)
```

From the documentation: "Wrapper around `query(android.net.Uri, String[], String, String[], String)` that gives the resulting Cursor to call `startManagingCursor(Cursor)` so that the activity will manage its lifecycle for you."

The first parameter is an URI and you should be already familiar with the URI concept. The second parameter is a projection that is a list of columns to return. A `SQL WHERE` clause can be passed as the third parameter, e.g.,

```
String where = People.NAME + " like '%mustermann%'";
```

To pass `selectionArgs` is also possible. For example, if you have a where statement like the following,

[7] See http://developer.android.com/guide/topics/providers/content-providers.html

```
… where Student.GENDER like ? and Student.AGE like ?;
```

which you may know as `PreparedStatement` as used in JDBC. The `selectionArgs` represent the list of arguments to replace each question mark in the statement. So, you can pass "female" for GENDER, and "24" for age. A `PreparedStatement` will be pre-compiled by the database and increases the performance especially in the case of more complex criteria because the database optimises, prepares, this kind of statements.

As a last parameter you can pass a sort order, i.e., `ASC` or `DESC`. It is reasonable to define a standard sort order in a constant, e.g.,

```
public static final String DEFAULT_SORT_ORDER = DATE+ " DESC";
```

Once returned you can iterate through the cursor. To do so, move the cursor to the first position

```
cur.moveToFirst()
```

In this case, it is just assumed that the query returns one contact only, of course this is not a good design, however, it is just to explain the concept. Later on you will have to deal with result sets returned by a query.

Now, to show the name and the number of the contact, you must access the columns of the row. It is important to know the data type of a column. Assumed, for example, the type of the first column is `String`, the method

```
cursor.getString(int columnIndex)
```

is required to access the column. As you can see, also the `columnIndex` is required. If you do not know the index of a column, which indeed is complicated if you access external CP, the method

```
getColumnIndexOrThrow(String ColumnName)
```

helps. If there exists a column with such a name, the index is returned, else wise an exception will be thrown.

Complete Example:

```
package de.rt.openx;

import android.app.Activity;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.provider.Contacts.People;
import android.widget.TextView;

public class OpenX extends Activity {
	/** Called when the activity is first created. */
	@Override
	public void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.openx);
		TextView tv_name = (TextView) findViewById(R.id.tv_name);
		TextView tv_mail = (TextView) findViewById(R.id.tv_mail);
```

```
            Uri contacts =  People.CONTENT_URI;

            Cursor cur= managedQuery(contacts, null, null, null,
            People.NAME +" ASC");
            if (cur.getCount() == 1) {
                  cur.moveToFirst();
                  tv_name.setText(cur.getString
                  (cur.getColumnIndexOrThrow(People.NAME)));
                  tv_mail.setText(cur.getString
                  (cur.getColumnIndexOrThrow(People.NUMBER)));
            }


            /*
             * Another variant if you have to deal with a result set rather
             * than a single entry.
             */

/*          String where = People.NAME + " like '%lessner%'";
            String[] projection = new String[] {
                People._ID,
                People.NAME,
                People.NUMBER
             };
            cur = managedQuery(contacts, projection, where, null,
            People.NAME +" ASC");
            cur.moveToFirst();
            do{
                  System.out.println(cur.getString(1));
                  System.out.println(cur.getString(2));
            }while (cur.moveToNext());
*/
      }
}
```

Binding

This section outlines how a CP is bound to an Activity.

As you see in the code example of the last section, Activity defines the method `managedQuery()`. As shown in figure 6 `query(...)` is also possible. The method `query(...)`, however, is defined in `ContentResolver` and bound to the Activity and the method `getContentResolver()` returns the corresponding `ContentResolver` , which in turn allows you to access the CP of an Activity.

If, however, the CP is bound via the method `managedQuery()`, the Activity has control over the cursor and the cursor is subjected to the life cycle of the Activity. Hence, if the Activity is closed the cursor is closed, too! By contrast, if the CP is bound by the method `getContentResolver().query()` the developer is responsible for the cursor.

An unmanaged cursor, i.e., not managed by the Activity, can become a managed one. Just pass the cursor to the method `startManagingCursor(Cursor c)`.

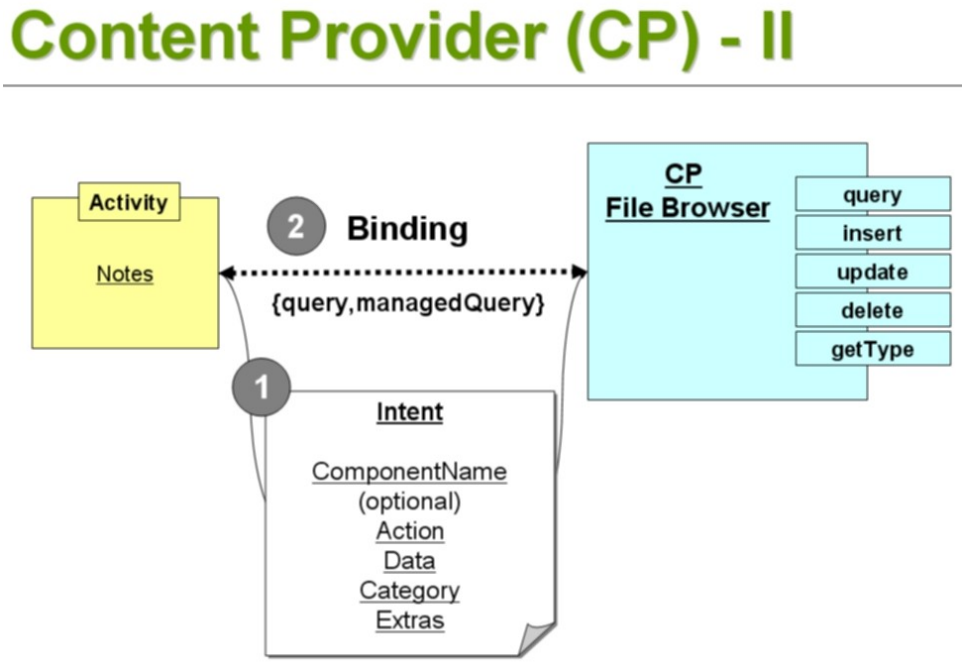

Illustration 6: CP and Binding

As an example consider the following code to update a note:

```
private boolean save() {
      try{
            ContentValues values = new ContentValues();
            values.put(Note._ID,cursor.getInt(
            cursor.getColumnIndexOrThrow(Note._ID)));

            values.put(Note.BODY,ed_text_body.getText().toString());
            values.put(Note.DATE,date_view.getText().toString());
```

```
            values.put(Note.LECTURE_FK,

            spin_lecture.getItemIdAtPosition(
                  spin_lecture.getSelectedItemPosition()));

            values.put(Note.TITLE,
                  ed_text_title.getText().toString());

            getContentResolver().update(getIntent().getData(),
                  values,
                  "",
                  null);
                  return true;
      }catch (Exception e){
            Log.e(TAG, e.getLocalizedMessage());
            return false;
      }
}
```

You may need this code later! So, do not forget that it is here.

Notepad Tutorial

This section is the actual exercise you will have to accomplish. As said in the beginning of this part, we provide parts of the application and your task is to finish the project which means, there will be a couple of gaps in the code you have to fill. First of all, import the project if not already done. The name of the project is **Notes-Student**.

The Tutorial – Preliminaries

In this section we introduce the example and step into some important details to give you a better understanding of how the application Notes-Student works. We do this on a class base and describe the class **Note, ListNote, and NotesProvider**. Your job is to finish the application with the help and information provided in this section as well as the Android Documentation, specifically the Notepad Tutorial[8], as already said in the section Goal.

Class Note

The class Note represents a note, is abstract and implements `BaseColumns`. Bear in mind during the whole tutorial that `BaseColumns` defines **`_ID`**.

```
public abstract class Note implements BaseColumns
```

Note is abstract because there is no real need to instantiate a note and the application will access the cursor instead, and we just pass the values of a note and not the entire object. Some may argue this is not a good way to program Java, and indeed, they are right -somehow-. But, the other side is that a mobile device does not have a comparable amount of RAM as your Home PC or even a server has. So, to directly work with cursors and just pass the data instead of complete objects reduces the amount of memory required.[9]

Nevertheless, to represent the structure of a note, class note defines all its attributes names as `String` values. This values are used, for example, to create the corresponding table or to access a cursor.

```
public static final String TITLE = "title";
public static final String LECTURE = "lecture";
public static final String LECTURER = "lecturer";
public static final String DATE = "creationdate";
public static final String BODY = "body";
public static final String USER = "user";
```

Since the CP provides access to notes a couple of other information is needed, like the default sort order or the authority URI. We do this all within this class.

```
public static final String DEFAULT_SORT_ORDER = DATE+ " DESC";
public static final String AUTHORITY = "de.rt.notes.notesprovider";
```

[8] http://developer.android.com/resources/tutorials/notepad/index.html (2010-11-13)

[9] *During the preparation of this workshop we developed a similar solution that is entirely based on objects. We further detached the objects and the Data Access was disconnected within the mobile application. Consequently, it would be ideal to measure and compare the performance of both approaches with respect to CPU load, memory usage, and eventually the battery life /future work).*

As already discussed, each CP should define a `CONTENT_URI` for each type that can be accessed. We define the `CONTENT_URI` for note also as a constant value in the class `Note` itself.

```
public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY
                + "/notes");
```

We further define a `CONTENT_TYPE` for a single note and for a set of notes. Please, just notice this issue here and we will come back to this issue later on.

```
public static final String CONTENT_TYPE =
"vnd.android.cursor.dir/vnd.notes.note";

public static final String CONTENT_ITEM_TYPE =
vnd.android.cursor.item/vnd.notes.note";
```

Class (ListActivity) ListNote

This Activity is the default Activity of the application and its purpose is to list all existing notes. Starting from this list, a user can either create, edit, delete, or modify a note. To edit or delete a note the user has to click on the note, to create a new note the user must use the menu.

In the following we go through the code step by step. You find also comments in the code as you may have noticed.

The class definition is as follows:

```
public class ListNote extends ListActivity
```

As you can see `ListNote` is a specialisation of `ListActivity` and not `Activity`. What does that mean? Beside the methods defined in Activity, a `ListActivity` has some specialised methods like `protected void onListItemClick(ListView l, View v, int position, long id)`, `public long getSelectedItemId()`, or `public int getSelectedItemPosition()`. These three methods, if they are overwritten properly, allow to easily handle a selection of one item in the list. The method `public long getSelectedItemId()` relies on the specification that an id of an entity is named `_ID` in your table and since the CP binds a returned result set to the `ListActivity`, there is nothing to do for a developer to get the id of the selected entity.

When the Activity is loaded, already existing notes should be displayed. To query existing notes requires to call the query method of the CP. To ensure that an URI is set for this Intent, the Intent is required and to set the URI, method `setData(...)` needs to be called. The default URI is necessary here because all notes should be displayed.

```
Intent intent = getIntent();
if (intent.getData() == null) {
      //DEFAULT URI
      intent.setData(Note.CONTENT_URI);
}
```

Now, it is getting a bit more complicated.

```
Cursor cursor = managedQuery(
getIntent().getData(),
NoteHelper.NOTEPROJECTION,
null,
null,
Note.DEFAULT_SORT_ORDER);
```

You should be already familiar with the method **`public final Cursor managedQuery (Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder).`**

Recall, as a first parameter the URI is passed, which is set in the Intent. The second parameter passed is the projection that is defined in the class NoteHelper as:

```
public static final String[] NOTEPROJECTION = new String[] {
        Note._ID, // 0, _ID see BaseColumns
        Note.TITLE,// 1, TITLE see Note
        Note.LECTURE,
        Note.LECTURER,
        Note.DATE,
        Note.BODY,
        Note.USER
      };
```

Since this query is without any criteria, we want to see all notes, the third, and fourth parameter are set to `null`. And since the notes should not be sorted in a specific way we just pass the default sort order as defined in the class `Note`.

How does the actual population of the list take place? The configuration `AndroidMainfest.xml` defines class `de.rt.notes.notesprovider` as CP.

```
<provider android:name="NotesProvider"
android:authorities="de.rt.notes.notesprovider" />
```

Each CP must implement the Interface `ContentProvider`, and thus, it ensures that each call to the `query(...)`, …, `delete(...)` method of a CP can be resolved. As already outlined in the last section, a call to `managedQuery(…)` implicitly leads to call the method `query()` of `NotesProvider`. Eventually, the returned cursor is bound to the `ListActivity`.

The following lines of code eventually wrap – adapt – the cursor to view the entries of a cursor.

```
SimpleCursorAdapter adapter = new SimpleCursorAdapter(
this,
R.layout.noteslist_item,
cursor,
new String[] {Note.TITLE},
new int[] {android.R.id.text1});
setListAdapter(adapter);
```

A `SimpleCursorAdapter` instance is used to display the data and the constructor of `SimpleCursorAdapter` is defined as follows:

```
public SimpleCursorAdapter (Context context, int layout, Cursor c,
String[] from, int[] to)
```

From the Android Documentation:
"An easy adapter to map columns from a cursor to TextViews or ImageViews defined in an XML file. You can specify which columns you want, which views you want to display the columns, and the XML file that defines the appearance of these views. Binding occurs in two phases."[10]

In other words:

Per each entry in the result set one entry is added to the `ListActivity` which is represented by the parameter `this`. To do so, also a `View` element is required, in this case, as defined in `noteslist_item.xml`.

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

<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/text1"
    android:layout_width="fill_parent"
    android:layout_height="?android:attr/listPreferredItemHeight"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:gravity="center_vertical"
    android:paddingLeft="5dip"
    android:singleLine="true"
/>
```

The adapter requires a projection too, but because we just want to show the title of a note we just pass `new String[] {Note.TITLE}` and not a multi-valued array.

The last parameter is defined as `int[] to` and we pass the index `new int[] {android.R.id.text1}`. Regarding this parameter the Android Documentation says:
"The views that should display column in the "from" parameter. These should all be TextViews. The first N views in this list are given the values of the first N columns in the from parameter. Can be null if the cursor is not available yet." [11]

According to this, per each column in the `ListView`, a `TextView` – Element is required. We are a bit lazy here and just pass the already defined `TextView` – Element `android.R.id.text1`.

Finally, the adapter is set:

```
setListAdapter(adapter);
```

Next, `ListNote` implements the method:

```
@Override
protected void onListItemClick(ListView l, View v, int position, long id){}
```

[10] http://developer.android.com/reference/android/widget/SimpleCursorAdapter.html
[11] http://developer.android.com/reference/android/widget/SimpleCursorAdapter.html

As you see, this method is preceded by the optional annotation `@Override`, and as you may know, as well as the annotation's name says, the method `onListItemClick(...)` as defined in `ListActivity` is overwritten here. The method is a listener method that fires if an entry is selected by the user.

By having a look into the implementation of the method, you see that the URI is set, an Activity is started, and a call to the super class takes place.

```
Uri uri = ContentUris.withAppendedId(getIntent().getData(), id);
startActivity(new Intent(Intent.ACTION_EDIT, uri));
super.onListItemClick(l, v, position, id);
```

The goal is: if an entry has been selected the entry should be passed to the Activity `NoteEdit`.

As you may have observed, it is possible to just use the id because Android postulates `_ID` as the name of the entities id, at least in the database and hence the projection.

As we have already discussed, too, the URI that refers to a single entity is of the following form

```
content://de.rt.notes.notesprovider/notes/<ID>
```

To construct such an URI the class `ContentUris` provides the static method `withAppendedId`, to append an `ID` as last part to an URI. So, what we expect the CP to do, is to query the database and return the correct note to modify or delete the note in `NoteEdit`.

The last thing that needs to be done is to fire the Intent.

```
startActivity(new Intent(Intent.ACTION_EDIT, uri));
```

As you can see, the parameter `Intent.ACTION_EDIT` is passed. Notice, this is an implicit Intent and if you have a look into the `AndroidManifest.xml` you will find:

```
<activity android:name=".NoteEdit" android:label="@string/app_name">
<intent-filter android:label="@string/resolve_edit">
      <action android:name="android.intent.action.VIEW" />
      <action android:name="android.intent.action.EDIT" />
      <action android:name="de.rt.note.note.action.EDIT_NOTE" />
      <category android:name="android.intent.category.DEFAULT" />
      <data android:mimeType="vnd.android.cursor.item/vnd.notes.note" />
</intent-filter>
</activity>
```

Based on this information the Android Framework is able to automatically route the Intent to `NoteEdit` if the URI `android.intent.action.EDIT` and `mimeType="vnd.android.cursor.item/vnd.notes.note"` match. The `mimeType` is defined in the CP.

We skip the remaining code and continue with the class **`NotesProvider`** at this stage.

Class NotesProvider

In this section we explain the important internals of the CP.

The class definition of `NotesProvider` is as follows:

```
public class NotesProvider extends ContentProvider
```

Since `NotesProvider` extends the abstract class `ContentProvider` the following abstract methods of `ContentProvider` must be implemented by `NotesProvider`. We have explained the reasons already.

- query(Uri, String[], String, String[], String)
- insert(Uri, ContentValues)
- update(Uri, ContentValues, String, String[])
- delete(Uri, String, String[])
- getType(Uri)

An implementation of these methods requires a data source, a database in our case. Notice, it is not specified what exactly a data source is, however, in most cases the data source will be an SQLite database as Android ships with SQLite.

First of all some constant values and attributes are required.

Constant values:

```
private static final String TAG = "NotesProvider";
```

Just a name.

```
private static final String DATABASE_NAME = "notes.db";
```

Database name.

```
private static final int DATABASE_VERSION = 2;
```

Database Version.

```
private static final String NOTES_TABLE_NAME = "notes";
```

Name for table notes.

```
private static final String LECTURE_TABLE_NAME = "lecture";
```

Name for the table lecture.

Attributes:

```
private static HashMap<String, String> notesProjectionMap;
```

Projection for notes.

Next, `NotesProvider` defines a static inner class which inherits from `SQLiteOpenHelper`, and therefore it has to implement the abstract methods `onCreate(…)` and `onUpgrade(…)`.

```
private static class DatabaseHelper extends SQLiteOpenHelper {
DatabaseHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
}

@Override
public void onCreate(SQLiteDatabase db) {
      db.execSQL("...;");
      db.execSQL("...;");
      …
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
      Log.w(TAG, "Upgrading database from version " + oldVersion + " to "
      + newVersion + ", which will destroy all old data");
      db.execSQL("DROP TABLE IF EXISTS " + NOTES_TABLE_NAME);
      onCreate(db);
      }
}
```

We are specifically interested in the method `onCreate(…)`. As you can see, there are some `db.execSQL` statements to create tables or to insert some sample data, for instance. Notice, however, these statements are executed only if the database does not exist yet. Thus, to re-create the database you will have to delete the database. To do so, connect to the running Android device and delete the directory with
`rm -R /data/data/de.rt.notes/databases`.
Each application has its own directory in `/data/data`.

After the inner class definition you see a class variable

```
private DatabaseHelper mOpenHelper;
```

of type `DatabaseHelper`, hence, an instance of the private inner class.

Not only DatabaseHelper, also `ContentProvider` has an `onCreate()` method, which needs to be implemented. This method is called each time the CP is instantiated, and thus, this is the correct time to instantiate the `DatabaseHelper` as well.

```
@Override
public boolean onCreate() {
      mOpenHelper = new DatabaseHelper(getContext());
      return true;
}
```

Notice, `getContext()` is implemented in `ContentProvider`, already.

Time to have a look into the method `query(…)`. If you are confused why the method has the same signature than `managedQuery(…)` please go back to the section where `managedQuery(…)` has been discussed.

```
@Override
public Cursor query(Uri uri, String[] projection, String selection,
                    String[] selectionArgs, String sortOrder) {
```

The first line of code in this method instantiates a `SQLiteQueryBuilder` object.

```
SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
```

As the name suggests, an object of type `SQLiteQueryBuilder` helps to construct a query. Please notice, it is also possible to execute any valid SQL statement, however, we focus on this specific way.

Regarding the next lines you find the following switch statement.

```
switch (sUriMatcher.match(uri)) {
case NOTES:
      qb.setTables(NOTES_TABLE_NAME);
      qb.setProjectionMap(notesProjectionMap);
      break;

case NOTE_ID:
      qb.setTables(NOTES_TABLE_NAME);
      qb.setProjectionMap(notesProjectionMap);
      qb.appendWhere(Note._ID + "=" + uri.getPathSegments().get(1));
      break;
...
```

The purpose of this switch statement is very important. Query may refer to different types, e.g., note, lecture, lecturer, and some queries may refer to a single data entry, whereas others, relate to a set of data. Already, at the beginning of this class you find a static code block that, among other things, initialises a `sUriMatcher`.

```
private static final UriMatcher sUriMatcher;
static {
      sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
      sUriMatcher.addURI(Note.AUTHORITY, "notes", NOTES);
      sUriMatcher.addURI(Note.AUTHORITY, "notes/#", NOTE_ID);

      …
}
```

Regarding the Android documentation an `UriMatcher` is[12] a: *"Utility class to aid in matching URIs in content providers."*

To parse an URI is a common task, not only in Android, and to ease this task Android provides this nice little helper to parse URIs and return true if a given URI matches. So, each URI must be known by the `UriMatcher`. You can add an URI with:

[12] http://developer.android.com/reference/android/content/UriMatcher.html , BEACHTEN SIE bitte das dort dargestellte sehr gute Beispiel.

```
public void addURI (String authority, String path, int code)
```

The `authority` parameter should be known already from the URI section. Parameter `path` refers to the specific object, `Note` in this case. The last parameter, `code`, is important because the code will be returned as a result for a matching URI. Therefore, the class defines the constant values:

```
private static final int NOTES = 1;
private static final int NOTE_ID = 2;
```

Coming back to the switch statement, if the URI refers to notes all notes should be returned that fulfil the where condition, if applied. In case the URI ends with an ID, the corresponding note should be returned only. The URI `content://de.rt.notes.notesprovider/notes` represents the first, and the URI `content://de.rt.notes.notesprovider/notes/<ID>` the second case.

After the URI has been resolved the `QueryBuilder` is used to construct the query. In case the URI refers to a single entity only, i.e., the last part is an ID, the method `uri.getPathSegments().get(1)` returns the ID.

The switch block is followed by the code to eventually query the database.

```
SQLiteDatabase db = mOpenHelper.getReadableDatabase();
Cursor c = qb.query(db, projection, selection, selectionArgs, null,
                        null, null);
```

Now you find `getType(Uri uri)` as next method. As briefly mentioned above, there are so called `MimeTypes` that are required because without this information it is not possible to handle implicit intents without a doubt. The URI allows for a routing of the Intent, but once delivered, we need a mechanism that ensures some type safety. In case the URI relates to notes, for example, the `MimeType` is set accordingly. Notice, the `getType(...)` method is called by the Android framework and there is no code within the application calling this method.

```
public String getType(Uri uri) {
 switch (sUriMatcher.match(uri)) {
      case NOTES:
      return Note.CONTENT_TYPE;
      case NOTE_ID:
      return Note.CONTENT_ITEM_TYPE;
      …
}
```

We skip the explanation for the methods `delete(…)`, `insert(…)`, and `update (…)`. Actually, to implement these methods is up to you in the next exercise.

Exercise 1

As you can see there are a couple of gaps in the code and your task is it now to implement these gaps. Neither Activity **`NoteEdit`** nor the methods to **`update(...)`**, **`delete(...)`**,

and **`insert(...)`** have been explained. Do not focus on lecturer or even media content at this stage.

General Advice:

- Use the Android Documentation and the Notepad Tutorial provided there
- Use the SQLite3 tool (adb shell, sqlite3 <path to the database>). Notice, that if you use the wrong path, i.e., not the path to your application's database a new database will be created at this location. Obviously you won't see anything then.
- It is recommended to set **breakpoints**, **debug**, and to check the **LogCat**
- **Easiest things first!**
- **Ask questions!**
- **Teamwork!**

Class `NotesProvider` - Open tasks

In total you find **9 open tasks** in this class, just have look into **`NotesProvider.java`** first and for each task you will find an outline how to tackle the task.

Basically, there are two ways to start:

1) Do not focus on the problem that there is no table lecture yet and implement the update, delete, and insert method first. Once you are done, normalise the data model and provide a table lecture. Adapt the code for update, delete, and insert afterwards.

2) Focus on the normalisation first, provide a table for lecture, and implement the update, delete, and insert method afterwards.

Class `NoteEdit` – Open tasks

In this class you find 4 open tasks in total. Again, just have a look into `NoteEdit.java` first.

Task 1 is to populate the spinner to select a lecture. You can already use the query method of the `NotesProvider`. The second task is to implement the call to the `NotesProvider`'s delete method. Task 3 and 4 are the calls to the `NotesProvider`'s save and insert method. Neither the delete, nor the save or insert method are implemented in `NotesProvider`, except you have done it already as you have been asked to do in the last section.

Exercise 2

Change the spinner to select a lecture and implement an ListActivity listLectures to do this instead. You could use a button, for instance, that starts listLectures. Use the method `startActivityForResult` and catch the result returned from `listLectures` in `NoteEdit`. Moreover, it would be nice to have a separate CP for lectures.

Exercise 3

This exercise treats with SQLite3. Should not take a long time.
Create two tables with a 1 to many relationship.

```
Table A{ID:int PK, Name:Text}
Table B{ID:int PK, Name: Text, FK_A:int FK}
```

Show that SQLite3 is not aware of foreign key relationships and that it is indeed possible to use a non-existing Primary Key from table A in table B.

What could you do to ensure that this cannot happen, i.e., use a non-existing Primary Key as Foreign Key?

Have a look at http://www.sqlite.org/faq.html#q22