UNIT – IV

Database Connectivity Using SQLite

➢ Working with Application Preferences

Many applications need a lightweight data storage mechanism called shared preferences for storing application state, simple user information, configuration options, and other such information.

Android provides a simple preferences system for storing primitive application data at the Activity level and preferences shared across all of an application's activities.You cannot share preferences outside of the package. Preferences are stored as groups of key/value pairs.The following data types are supported as preference settings:

1. Boolean values
2. Float values
3. Integer values
4. Long values
5. String values

Preference functionality can be found in the SharedPreferences interface of the android.content package.To add preferences support to your application, you must take the following steps:

1. Retrieve an instance of a SharedPreferences object.
2. Create a SharedPreferences.Editor to modify preference content.
3. Make changes to the preferences using the Editor.
4. Commit your changes.

➢ Creating Private and Shared Preferences

Individual activities can have their own private preferences.These preferences are for the specific Activity only and are not shared with other activities within the application.The activity gets only one group of private preferences.

The following code retrieves the activity's private preferences:

```
import android.content.SharedPreferences;
...
SharedPreferences settingsActivity getPreferences(MODE_PRIVATE);
```

➢ Searching and Reading Preferences

Reading preferences is straightforward. Simply retrieve the SharedPreferences instance you want to read.You can check for a preference by name, retrieve strongly typed preferences, and register to listen for changes to the preferences.Following method describes some helpful methods in the SharedPreferences interface.

Method	Purpose
SharedPreferences.contains()	Sees whether a specific preference exists by name
SharedPreferences.edit()	Retrieves the editor to change these preferences
SharedPreferences.getAll()	Retrieves a map of all preference key/value pairs
SharedPreferences.getBoolean()	Retrieves a specific Boolean-type preference by name
SharedPreferences.getFloat()	Retrieves a specific Float-type preference by name
SharedPreferences.getInt()	Retrieves a specific Integer-type preference by name
SharedPreferences.getLong()	Retrieves a specific Long-type preference by name
SharedPreferences.getString()	Retrieves a specific String-type preference by name

➢ Adding, Updating, and Deleting Preferences

To change preferences, you need to open the preference Editor, make your changes, and commit them.Table 10.2 describes some helpful methods in the SharedPreferences.Editor interface

Method	Purpose
SharedPreferences.Editor.clear()	Removes all preferences. This operation happens first, regardless of when it is called within an editing session; then all other changes are made and
SharedPreferences.Editor.remove()	Removes a specific preference by name. This operation happens first, regardless of when it is called within an editing session; then all other changes are made and committed.
SharedPreferences.Editor.putBoolean()	Sets a specific Boolean-type preference by name.
SharedPreferences.Editor.putFloat()	Sets a specific Float-type preference by name.
SharedPreferences.Editor.putInt()	Sets a specific Integer-type preference by

	name.

➢ Storing Structured Data Using SQLite Databases

The Android SDK includes a number of useful SQLite database management classes.
Many of these classes are found in the android.database.sqlite package. Here you can find utility classes for managing database creation and versioning, database management, and query builder helper classes to help you format proper SQL statements and queries.
The package also includes specialized Cursor objects for iterating query results.You can also find all the specialized exceptions associated with SQLite.
Here we focus on creating databases within our Android applications. For that,we use the built-in SQLite support to programmatically create and use a SQLite database to store

SQLite is a opensource SQL database that stores data to a text file on a device. Android comes in with built in SQLite database implementation.

SQLite supports all the relational datbase features. In order to access this database , you don't need to establish any kind of connections for it like JDBC,ODBC e.t.c

➢ Creating a SQLite Database Instance Using the Application Context

The simplest way to create a new SQLiteDatabase instance for your application is to use the openOrCreateDatabase() method of your application Context, like this:

1)

```
import android.database.sqlite.SQLiteDatabase;
...
SQLiteDatabase mDatabase;
mDatabase openOrCreateDatabase(
“my_sqlite_database.db”,
SQLiteDatabase.CREATE_IF_NECESSARY, null);
```

2)

```
SQLiteDatabse mydatabase = openOrCreateDatabase("your database
name",MODE_PRIVATE,null);
```

➢ Finding the Application's Database File on the Device File System

/data/data/com.androidbook.SimpleDatabase/databases/my_sqlite_database.db

➢ Configuring the SQLite Database Properties

Now that you have a valid SQLiteDatabase instance, it's time to configure it. Some important database configuration options include version, locale, and the thread-safe locking feature.

```
import java.util.Locale;
...
mDatabase.setLocale(Locale.getDefault());
mDatabase.setLockingEnabled(true);

mDatabase.setVersion(1);
```

➢ Creating Tables and Other SQLite Schema Objects

Creating tables and other SQLite schema objects is as simple as forming proper SQLite statements and executing them.The following is a valid CREATE TABLE SQL statement.
This statement creates a table called tbl_authors.The table has three fields: a unique id number, which auto-increments with each record and acts as our primary key, and firstname and lastname text fields:

1)

```
CREATE TABLE tbl_authors (
id INTEGER PRIMARY KEY AUTOINCREMENT,
firstname TEXT, lastname TEXT);
```

encapsulate this CREATE TABLE SQL statement in a static final String variable (called CREATE_AUTHOR_TABLE) and then execute it on your database using the execSQL() method:

```
mDatabase.execSQL(CREATE_AUTHOR_TABLE);
```

2)

```
mydatabase.execSQL("CREATE TABLE IF NOT EXISTS TutorialsPoint(Username
VARCHAR,Password VARCHAR);");
mydatabase.execSQL("INSERT INTO TutorialsPoint VALUES('admin','admin');");
```

execSQL(String sql Object[] bindArgs)

This method not only insert data but also used to update or modify already existing data in database using bind arguments

➢ Creating and updating database with SQLiteOpenHelper

To create and upgrade a database in your Android application you create a subclass of the SQLiteOpenHelper class. In the constructor of your subclass you call the super() method of SQLiteOpenHelper, specifying the database name and the current database version.

In this class you need to override the following methods to create and update your database.

- onCreate() - is called by the framework, if the database is accessed but not yet created.
- onUpgrade() - called, if the database version is increased in your application code. This method allows you to update an existing database schema or to drop the existing database and recreate it via the onCreate() method.

Both methods receive an SQLiteDatabase object as parameter which is the Java representation of the database.

The `SQLiteOpenHelper` class provides the `getReadableDatabase()` and `getWriteableDatabase()` methods to get access to an `SQLiteDatabase` object; either in read or write mode.

The database tables should use the identifier `_id` for the primary key of the table. Several Android functions rely on this standard.

➢ SQLiteDatabase

SQLiteDatabase is the base class for working with a SQLite database in Android and provides methods to open, query, update and close the database.
More specifically SQLiteDatabase provides the insert(), update() and delete() methods.
In addition it provides the execSQL() method, which allows to execute an SQL statement directly.
The object ContentValues allows to define key/values. The *key* represents the table column identifier and the *value* represents the content for the table record in this column. ContentValues can be used for inserts and updates of database entries.
Queries can be created via the rawQuery() and query() methods or via the SQLiteQueryBuilder class .
rawQuery() directly accepts an SQL select statement as input.
query() provides a structured interface for specifying the SQL query.
SQLiteQueryBuilder is a convenience class that helps to build SQL queries.

➢ Inserting Records

We use the insert() method to add new data to our tables.We use the ContentValues object to pair the column names to the column values for the record we want to insert.

For example, here we insert a record into tbl_authors for J.K. Rowling:
import android.content.ContentValues;
...
ContentValues values new ContentValues();
values.put("firstname", "J.K.");
values.put("lastname", "Rowling");
long newAuthorID mDatabase.insert("tbl_authors", null, values);
The insert() method returns the id of the newly created record.We use this author id

to create book records for this author.

The addContact() method accepts Contact object as parameter. We need to build ContentValues parameters using Contact object. Once we inserted data in database we need to close the database connection.

addContact()

```
    // Adding new contact
publicvoidaddContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();

    ContentValues values = newContentValues();
    values.put(KEY_NAME, contact.getName()); // Contact Name
    values.put(KEY_PH_NO, contact.getPhoneNumber()); // Contact Phone Number

    // Inserting Row
    db.insert(TABLE_CONTACTS, null, values);
    db.close(); // Closing database connection
}
```

➢ Updating Records

You can modify records in the database using the update() method.The update() method takes four arguments:

- ✓ The table to update records
- ✓ A ContentValues object with the modified fields to update
- ✓ An optional WHERE clause, in which ? identifies a WHERE clause argument
- ✓ An array of WHERE clause arguments, each of which is substituted in place of the ?'s from the second parameter

Passing null to the WHERE clause modifies all records within the table, which can be useful for making sweeping changes to your database.

Most of the time, we want to modify individual records by their unique identifier.
The following function takes two parameters: an updated book title and a bookId.We find the record in the table called tbl_books that corresponds with the id and update that book's title. Again, we use the ContentValues object to bind our column names to our data values:

```
public void updateBookTitle(Integer bookId, String newtitle) {
ContentValues values new ContentValues();
values.put("title", newtitle);
mDatabase.update("tbl_books",
values, "id ?", new String[] { bookId.toString() });
}
```

Because we are not updating the other fields,we do not need to include them in the

- ✓ ContentValues object.We include only the title field because it is the only field we change.

updateContact() will update single contact in database. This method accepts Contact class object as parameter.

updateContact()

```
    // Updating single contact
publicintupdateContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();

    ContentValues values = newContentValues();
    values.put(KEY_NAME, contact.getName());
    values.put(KEY_PH_NO, contact.getPhoneNumber());

    // updating row
    returndb.update(TABLE_CONTACTS, values, KEY_ID + " = ?",
            newString[] { String.valueOf(contact.getID()) });
}
```

➢ Deleting Records

You can remove records from the database using the remove() method.The remove() method takes three arguments:

- ✓ The table to delete the record from
- ✓ An optional WHERE clause, in which ? identifies a WHERE clause argument
- ✓ An array of WHERE clause arguments, each of which is substituted in place of the ?'s from the second parameter

Passing null to the WHERE clause deletes all records within the table. For example, this function call deletes all records within the table called tbl_authors:

mDatabase.delete("tbl_authors", null, null);

Most of the time, though,we want to delete individual records by their unique identifiers.

The following function takes a parameter bookId and deletes the record corresponding to that unique id (primary key) within the table called tbl_books:

public void deleteBook(Integer bookId) {
mDatabase.delete("tbl_books", "id ?",
new String[] { bookId.toString() });

}

deleteContact() will delete single contact from database.

deleteContact()

```
    // Deleting single contact
publicvoiddeleteContact(Contact contact) {
    SQLiteDatabase db = this.getWritableDatabase();
    db.delete(TABLE_CONTACTS, KEY_ID + " = ?",
            newString[] { String.valueOf(contact.getID()) });
    db.close();
}
```

➢ Working with Transactions

Often you have multiple database operations you want to happen all together or not at all.
You can use SQL Transactions to group operations together; if any of the operations fails, you can handle the error and either recover or roll back all operations. If the operations all succeed, you can then commit them. Here we have the basic structure for a transaction:

mDatabase.beginTransaction();
try {
// Insert some records, updated others, delete a few
// Do whatever you need to do as a unit, then commit it
mDatabase.setTransactionSuccessful();
} catch (Exception e) {
// Transaction failed. Failed! Do something here.
// It's up to you.
} finally {
mDatabase.endTransaction();

}

➢ Working with Cursors

When results are returned from a SQL query, you often access them using a Cursor found in the android.database.Cursor class. Cursor objects are rather like file pointers; they allow random access to query results.

You can think of query results as a table, in which each row corresponds to a returned record.The Cursor object includes helpful methods for determining how many results were returned by the query the Cursor represents and methods for determining the column names (fields) for each returned record.The columns in the query results are defined by the query, not necessarily by the database columns.These might include calculated columns, column aliases, and composite columns.

Cursor objects are generally kept around for a time. If you do something simple (such as get a count of records or in cases when you know you retrieved only a single simple record), you can execute your query and quickly extract what you need; don't forget to close the Cursor when you're done, as shown here:

```
// SIMPLE QUERY: select * from tbl_books
Cursor c mDatabase.query("tbl_books",null,null,null,null,null,null);
// Do something quick with the Cursor here...

c.close();
```

The following method getContact() will read single contact row. It accepts id as parameter and will return the matched row from the database.

getContact()

```
    // Getting single contact
publicContact getContact(intid) {
    SQLiteDatabase db = this.getReadableDatabase();

    Cursor cursor = db.query(TABLE_CONTACTS, newString[] { KEY_ID,
            KEY_NAME, KEY_PH_NO }, KEY_ID + "=?",
            newString[] { String.valueOf(id) }, null, null, null, null);
    if(cursor != null)
        cursor.moveToFirst();

    Contact contact = newContact(Integer.parseInt(cursor.getString(0)),
            cursor.getString(1), cursor.getString(2));
    // return contact
    returncontact;
}
```

➢ Executing Simple Queries

Your first stop for database queries should be the query() methods available in the SQLiteDatabase class.This method queries the database and returns any results as in a

Cursor object

```
return database.query(DATABASE_TABLE,

new String[] { KEY_ROWID, KEY_CATEGORY, KEY_SUMMARY, KEY_DESCRIPTION
},

  null, null, null, null, null);
```

Parameters of the query() method

Parameter	Comment
String dbName	The table name to compile the query against.
String[] columnNames	A list of which table columns to return. Passing "null" will return all columns.
String whereClause	Where-clause, i.e. filter for the selection of data, null will select all data.
String[] selectionArgs	You may include ?s in the "whereClause"". These placeholders will get replaced by the values from the selectionArgs array.
String[] groupBy	A filter declaring how to group rows, null will cause the rows to not be grouped.
String[] having	Filter for the groups, null means no filter.
String[] orderBy	Table columns which will be used to order the data, null means no ordering.

1)

we called the query() method with only one parameter set to the table name.

```
Cursor c mDatabase.query(“tbl_books”,null,null,null,null,null,null);
```

This is equivalent to the SQL query

2)

```
SELECT * FROM tbl_books;
```

Add a WHERE clause to your query, so you can retrieve one record at a time:

```
Cursor c mDatabase.query(“tbl_books”, null, “id ?”, new String[]{“9”}, null, null, null);
```

This is equivalent to the SQL query

SELECT * tbl_books WHERE id 9;

3)

For example,
if you need only the titles of each book in the book table, you might use the following call to the query() method:

String asColumnsToReturn[] { “title”, “id” };
String strSortOrder “title ASC”;
Cursor c mDatabase.query(“tbl_books”, asColumnsToReturn, null, null, null, null, strSortOrder);

This is equivalent to the SQL query

SELECT title, id FROM tbl_books ORDER BY title ASC;

➢ Deleting Tables and Other SQLite Objects

You delete tables and other SQLite objects in exactly the same way you create them. Format the appropriate SQLite statements and execute them. For example, to drop our tables and triggers,we can execute three SQL statements:

mDatabase.execSQL(“DROP TABLE tbl_books;”);
mDatabase.execSQL(“DROP TABLE tbl_authors;”);

mDatabase.execSQL(“DROP TRIGGER IF EXISTS fk_insert_book;”);

➢ Keeping Track of Database Field Names

To make a class to encapsulate your database schema in a class, such as PetDatabase, shown here:

```
import android.provider.BaseColumns;
public final class PetDatabase {
private PetDatabase() {}
public static final class Pets implements BaseColumns {
private Pets() {}
public static final String PETS_TABLE_NAME ”table_pets”;
public static final String PET_NAME ”pet_name”;
public static final String PET_TYPE_ID ”pet_type_id”;
public static final String DEFAULT_SORT_ORDER ”pet_name ASC”;
}
```

```
public static final class PetType implements BaseColumns {
private PetType() {}
public static final String PETTYPE_TABLE_NAME ”table_pettypes”;
public static final String PET_TYPE_NAME ”pet_type”;
public static final String DEFAULT_SORT_ORDER ”pet_type ASC”;
}

}
```

➢ Extending the SQLiteOpenHelper Class

To extend the SQLiteOpenHelper class,we must implement several important methods, which help manage the database versioning.The methods to override are onCreate() , onUpgrade(), and onOpen().We use our newly defined PetDatabase class to generate appropriate SQL statements, as shown here:

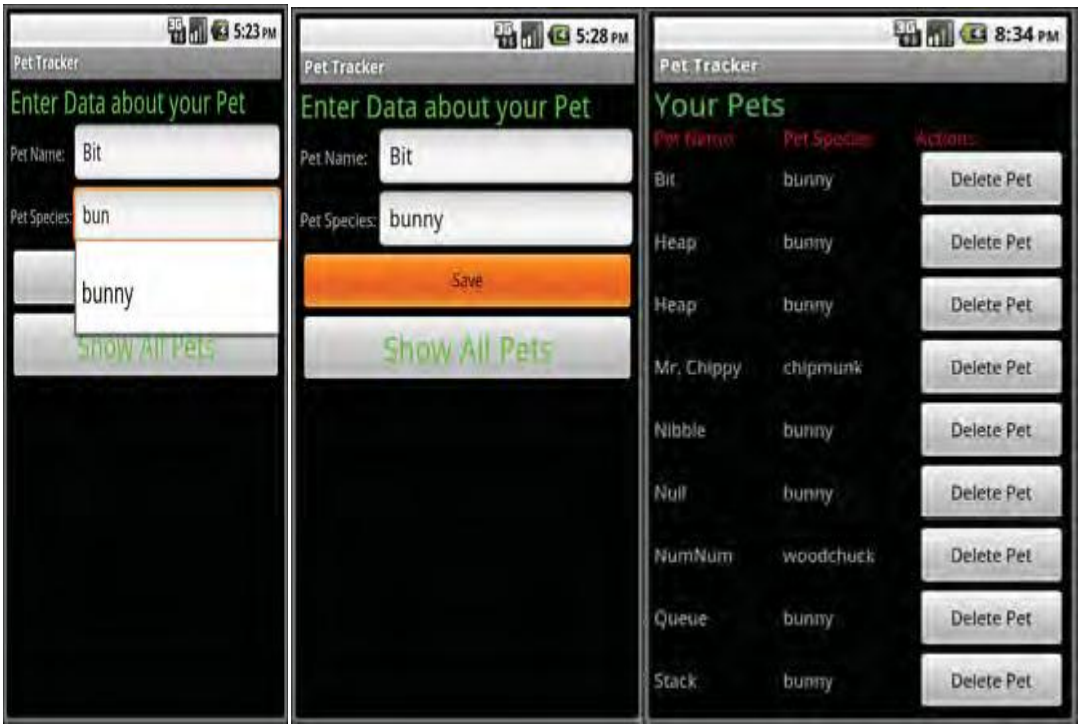
```
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import com.androidbook.PetTracker.PetDatabase.PetType;
import com.androidbook.PetTracker.PetDatabase.Pets;

class PetTrackerDatabaseHelper extends SQLiteOpenHelper {

private static final String DATABASE_NAME “pet_tracker.db”;
private static final int DATABASE_VERSION 1;
PetTrackerDatabaseHelper(Context context) {
super(context, DATABASE_NAME, null, DATABASE_VERSION);
}
@Override
public void onCreate(SQLiteDatabase db) {
db.execSQL(“CREATE TABLE “ +PetType.PETTYPE_TABLE_NAME+” (“
+ PetType._ID + “ INTEGER PRIMARY KEY AUTOINCREMENT ,”
+ PetType.PET_TYPE_NAME + “ TEXT”
+ “);”);
db.execSQL(“CREATE TABLE “ + Pets.PETS_TABLE_NAME + “ (“
+ Pets._ID + “ INTEGER PRIMARY KEY AUTOINCREMENT ,”
+ Pets.PET_NAME + “ TEXT,”
+ Pets.PET_TYPE_ID + “ INTEGER” // FK to pet type table
+ “);”);
}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
int newVersion){
// Housekeeping here.
```

```
// Implement how “move” your application data
// during an upgrade of schema versions
// Move or delete data as required. Your call.
}
@Override
public void onOpen(SQLiteDatabase db) {
super.onOpen(db);
}

}
```



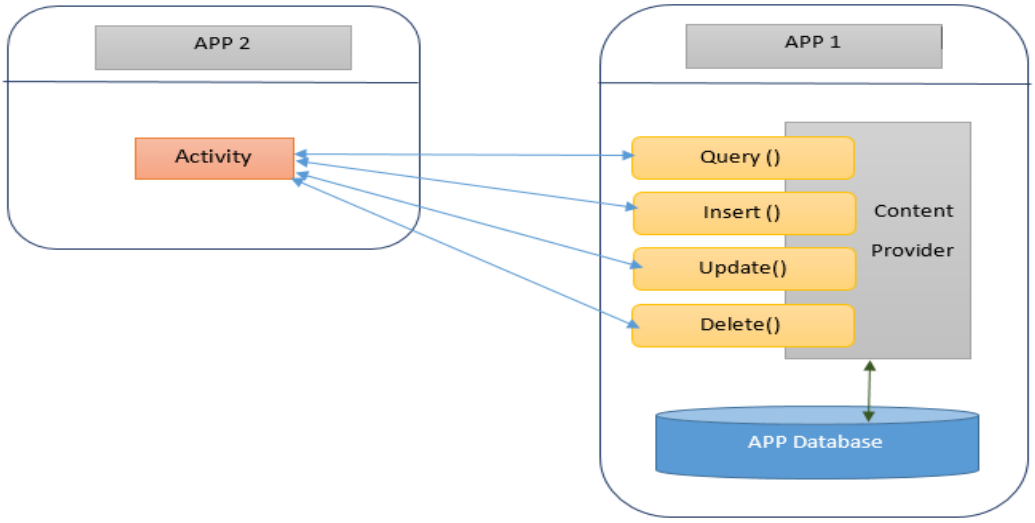
Content provider with example

In Android security model, one application cannot directly access (read/write) other application's data. Every application has its own id data directory and own protected memory area.

Content provider is the best way to share data across applications. Content provider is a set of data wrapped up in a custom API to read and write. Applications/Processes have to register themselves as a provider of data. Other applications can request Android to read/write that

data through a fixed API. Content provider API adheres to CRUD principle.

Examples for content provider are Contacts-that exposes user information to other applications, Media store-Allows other applications to access,store media files.



Above diagram shows how content provider works. App 1 stores its data in its own database and provides a provider. App 2 communicates with the provider to access App 1's data.

Content providers are simple interfaces which uses standard insert(), query(), update(), delete() methods to access application data. So it is easy to implement a content provider.

A special URI starting with ***content://*** will be assigned to each content providers and that will be recognized across applications.

Writing a content provider:

The ContentProvider class is the central component of a content provider. To create a content provider we have to

1.Create sub class for ContentProvider.

2. Define content URI
3. Implement all the unimplemented methods. insert(), update(), query(), delete(), getType().
4. Declare the content provider in AndroidManifest.xml

Defining URI:

Content provider URI consists of four parts.

content://authority/path/id
content:// All the content provider URIs should start with this value
'authority' is Java namespace of the content provider implementation. (fully qualified Java package name)

'path' is the virtual directory within the provider that identifies the kind of data being requested.
'id' is optional part that specifies the primary key of a record being requested. We can omit this part to request all records.

Adding new records:

We need to override insert() method of the ContentProvider to insert new record to the database via content provider. The caller method will have to specify the content provider URI and the values to be inserted without the ID. Successful insert operation will return the URI value with the newly inserted ID.
For example: If we insert a new record to the content provider content://com.example/sample the insert method will return content://com.example/sample/1

Updating records:

To update one or more records via content provider we need to specify the content provider URI. update() method of the ContentProvider is used to update records. We have to specify the ID of the record to update a single record. To update multiple records, we have to specify 'selection' parameter to indicate which rows are to be changed.
This method will return the number of rows updated.

Deleting records:

Deleting one or more records is similar to update process. We need to specify either ID or 'selection' to delete records. delete() method of the ContentProvider will return the number of records deleted.

Querying the content provider:

To query the data via content provider we orverride query() method of ContentProvider. This method has many parameters. We can specify list columns to put into the result cursor using 'projection' parameter. We can specify 'selection' criteria. We can specify 'sortOrder' too.

If we do not specify projection, all the columns will be included in the result cursor. If we do not specify sortOrder the provider will choose its own sort order.

getType() method:

This method is used handle requests for the MIME type of the data at the given URI. We use either vnd.android.cursor.item or vnd.android.cursor.dir/ vnd.android.cursor.item is used to represent specific item. Another one is used to specify all items.

Registering the provider in AndroidManifest.xml

As with any other major parts of Android application, we have to register the content providers too in the AndroidManifest.xml. <provider> element is used to register the content providers. <application> is its parent element.

```
<provider
android:name=".MyProvider"android:authorities="com.example.conte
ntproviderexample.MyProvider">
</provider>
```

Here authorities is the URI authority to access the content provider. Typically this will be the fully qualified name of the content provider class.

- A content provider is implemented as a subclass of **ContentProvider** class and must implement a standard set of APIs that enable other applications to perform transactions.

```
publicclassMyContentProviderextendsContentProvider{

}
```

- **onCreate()** This method is called when the provider is started.
- **query()** This method receives a request from a client. The result is returned as a Cursor object.

- **insert()**This method inserts a new record into the content provider.
- **delete()** This method deletes an existing record from the content provider.
- **update()** This method updates an existing record from the content provider.
- **getType()** This method returns the MIME type of the data at the given URI.

Content URIs

```
<prefix>://<authority>/<data_type>/<id>
```

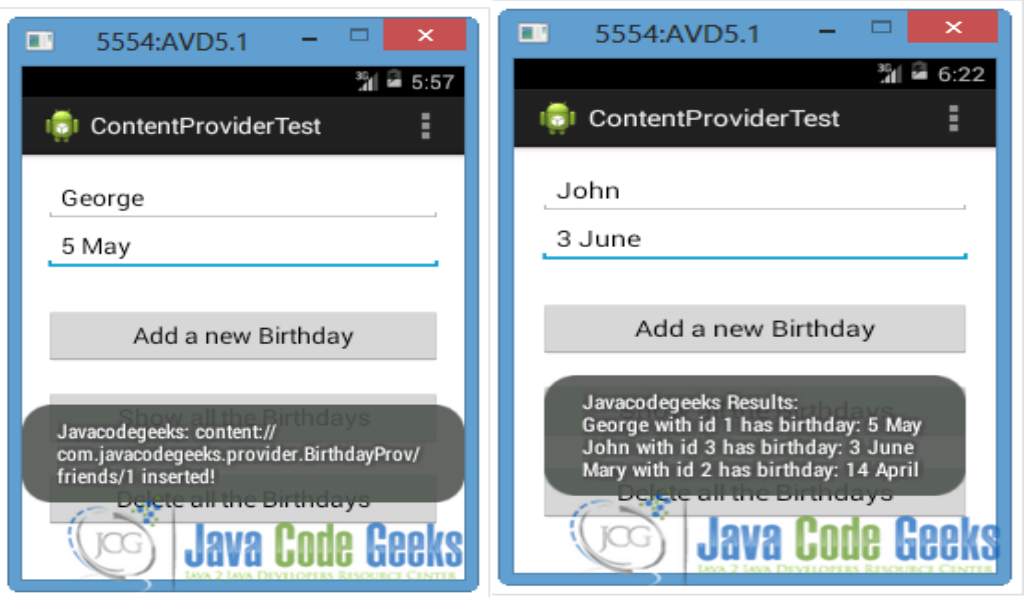
Part	Description
prefix	This is always set to content://
authority	This specifies the name of the content provider, for example *contacts*, *browser* etc. For third-party content providers, this could be the fully qualified name, such as *com.tutorialspoint.statusprovider*
data_type	This indicates the type of data that this particular provider provides. For example, if you are getting all the contacts from the *Contacts* content provider, then the data path would be *people* and URI would look like this *content://contacts/people*
id	This specifies the specific record requested. For example, if you are looking for contact number 5 in the Contacts content provider then URI would look like this *content://contacts/people/5*.

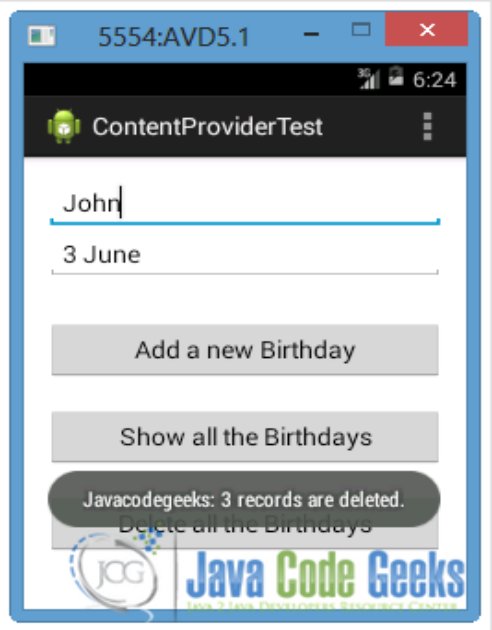
Sample application:

In this sample, we are going to create two application. First application will share its data through content provider and the second application will access the data.

1. Application 'MyProvider'

This application will create a content provider and share its data. This uses SQLite database 'mydb'. It has a table called 'names'. The table names have two columns id,name.
The URI of this content provider is
content://com.example.contentproviderexample.MyProvider/cte
We have layout to insert new records to the database table.





Code

1) Open res/values/strings.xml file to the "strings.xml" tab and paste the following code.

```
<?xmlversion="1.0"encoding="utf-8"?>
<resources>
<stringname="hello">Hello World, Contentex3Activity!</string>


<stringname="app_name">ContentProviderTest</string>
```

```
    <stringname="action_settings">Settings</string>
    <stringname="birthday">Birthday</string>
    <stringname="name">Name</string>
    <stringname="add">Add a new Birthday</string>
    <stringname="show">Show all Birthdays</string>
    <stringname="delete">Delete all Birthdays</string>
    <dimenname="activity_vertical_margin">12pt</dimen>
    <dimenname="activity_horizontal_margin">12pt</dimen>
    <stringname="BirthProvider">BirthProvider</string>
</resources>
```

2) Main.xml

```
    <?xmlversion="1.0"encoding="utf-8"?>
    <RelativeLayoutxmlns:android="http://schemas.android.com/apk/res/
    android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        tools:context=".MainActivity">

        <EditText
        android:id="@+id/name"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="@string/name"/>

        <EditText
        android:id="@+id/birthday"
        android:layout_width="fill_parent"
    android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/name"
        android:layout_below="@+id/name"
        android:ems="10"
        android:hint="@string/birthday"/>

        <Button
        android:id="@+id/btnAdd"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
```

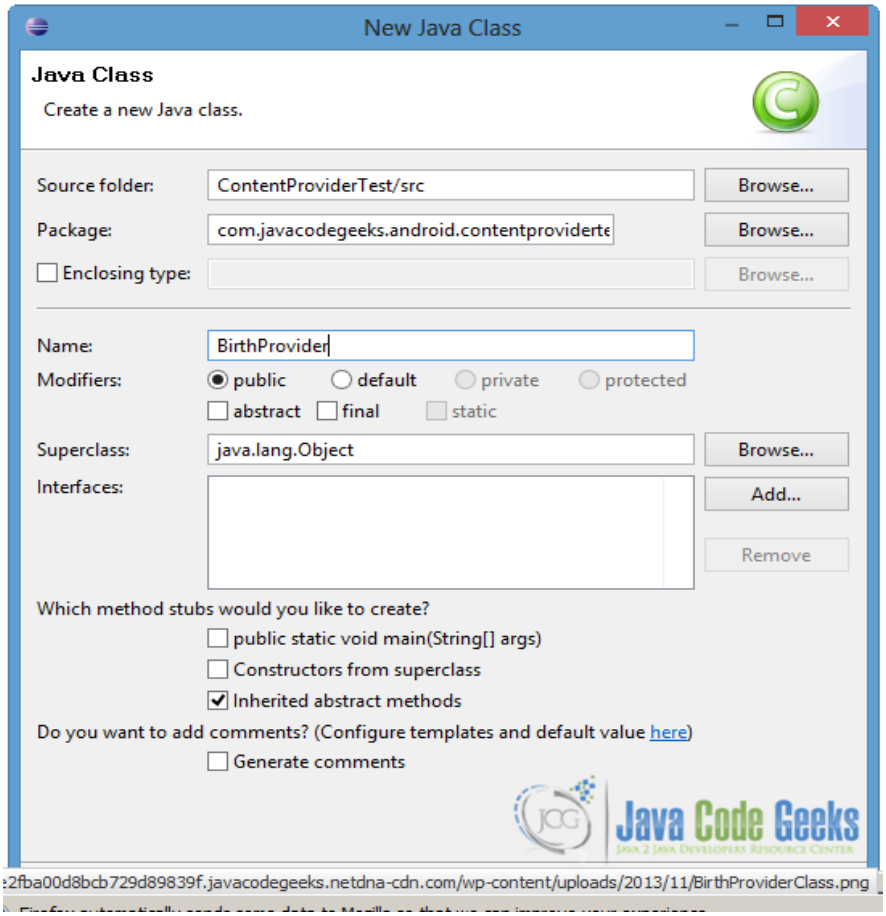
```
        android:onClick="addBirthday"
        android:layout_alignLeft="@+id/birthday"
        android:layout_below="@+id/birthday"
        android:layout_marginTop="30dp"
        android:text="@string/add"/>

        <Button
        android:id="@+id/btnShow"
    android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/btnAdd"
        android:layout_below="@+id/btnAdd"
        android:layout_marginTop="20dp"
        android:onClick="showAllBirthdays"
        android:text="@string/show"/>

        <Button
        android:id="@+id/btnDelete"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_below="@+id/btnShow"
        android:layout_alignLeft="@+id/btnShow"
        android:layout_marginTop="20dp"
        android:onClick="deleteAllBirthdays"
        android:text="@string/delete"/>

    </RelativeLayout>
```

3) Right click to com.androidbook.contentex3 package → New → Class.
Specify the name for the new Class and the package you want to put it. We will name it BirthProvider and we will put it in the same package as the MainActivity.java file.



BirthProvider is a subclass of ContentProvider and in order to identify the data in the provider we should use Content URIs. A Content URI has the following format:

```
1 content://<authority>/<path>
```

At the <authority> part we specify the name of the provider (in our occasion is the com.javacodegeeks.provider.Birthday, as you can see in the code below) and the <path> part points to the table. If a specific record is requested, we will add the <id> of that record at the end of the Content URI.

As we already mentioned we are going to use SQLite database, so in order to create and manage it we will use SQLiteOpenHelper. Also, we are going to use theUriMatcher Class, which maps the content URIs with particular patterns. That will help us to choose the desired action for an incoming content URI.

In addition, we want our content provider to work properly so we need to override the functions onCreate(), query(), insert(), update(), delete(), getType(). To learn more about these functions, you can visit the ContentProvider Class Overview.

Open `src/com.javacodegeeks.android.contentprovidertest/BirthProvider.java` and paste the following code:

```
package com.androidbook.contentex3;

import java.util.HashMap;

import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.Context;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
importandroid.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteQueryBuilder;
import android.net.Uri;
import android.text.TextUtils;
import android.util.Log;

publicclass BirthProvider extends ContentProvider{
      // fields for my content provide
staticfinal String PROVIDER_NAME =
"com.androidbook.contentex3.birthday";
staticfinal String URL = "content://" + PROVIDER_NAME + "/friends";
staticfinal Uri CONTENT_URI = Uri.parse(URL);

// fields for the database
staticfinal String ID = "id";
staticfinal String NAME = "name";
staticfinal String BIRTHDAY = "birthday";

// integer values used in content URI
staticfinalintFRIENDS = 1;
staticfinalintFRIENDS_ID = 2;

DBHelper dbHelper;

// projection map for a query
privatestatic HashMap<String, String>BirthMap;
```

```
// maps content URI "patterns" to the integer values that were set
above
staticfinal UriMatcher uriMatcher;
static{
uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
uriMatcher.addURI(PROVIDER_NAME, "friends", FRIENDS);
uriMatcher.addURI(PROVIDER_NAME, "friends/#", FRIENDS_ID);
                  }

// database declarations
private SQLiteDatabase database;
staticfinal String DATABASE_NAME = "BirthdayReminder";
staticfinal String TABLE_NAME = "birthTable";
staticfinalintDATABASE_VERSION = 1;
staticfinal String CREATE_TABLE =
" CREATE TABLE " + TABLE_NAME +
" (id INTEGER PRIMARY KEY AUTOINCREMENT, " +
" name TEXT NOT NULL, " +
" birthday TEXT NOT NULL);";

  // class that creates and manages the provider's database
privatestaticclass DBHelper extends SQLiteOpenHelper{

      public DBHelper(Context context) {
super(context, DATABASE_NAME, null, DATABASE_VERSION);
      // TODO Auto-generated constructor stub
                  }

@Override
publicvoid onCreate(SQLiteDatabase db) {
// TODO Auto-generated method stub
      db.execSQL(CREATE_TABLE);
}

@Override
publicvoid onUpgrade(SQLiteDatabase db, int oldVersion,
            int newVersion) {
// TODO Auto-generated method stub
      Log.w(DBHelper.class.getName(),
            "Upgrading database from version " + oldVersion + " to "
               + newVersion + ". Old data will be destroyed");
               db.execSQL("DROP TABLE IF EXISTS " +  TABLE_NAME);
         onCreate(db);
}
```

```
 }
@Override
publicint delete(Uri uri, String selection, String[] selectionArgs) {
           // TODO Auto-generated method stub
int count = 0;

switch (uriMatcher.match(uri)){
caseFRIENDS:
// delete all the records of the table
          count = database.delete(TABLE_NAME, selection,
selectionArgs);
break;
caseFRIENDS_ID:
       String id = uri.getLastPathSegment(); //gets the id
       count = database.delete( TABLE_NAME, ID +  " = " + id +
                                  (!TextUtils.isEmpty(selection) ? "
                AND (" +selection + ')' : ""), selectionArgs);
break;
default:
thrownew IllegalArgumentException("Unsupported URI " + uri);
                           }


getContext().getContentResolver().notifyChange(uri, null);
                return count;
     }

@Override
public String getType(Uri uri) {
// TODO Auto-generated method stub
switch (uriMatcher.match(uri)){
// Get all friend-birthday records
caseFRIENDS:
return"vnd.android.cursor.dir/vnd.example.friends";
     // Get a particular friend
caseFRIENDS_ID:
return"vnd.android.cursor.item/vnd.example.friends";
                default:
thrownew IllegalArgumentException("Unsupported URI: " + uri);
                           }
     }

@Override
public Uri insert(Uri uri, ContentValues values) {
     // TODO Auto-generated method stub
```

```
long row = database.insert(TABLE_NAME, "", values);

      // If record is added successfully
if(row > 0) {
            Uri newUri = ContentUris.withAppendedId(CONTENT_URI, row);

      getContext().getContentResolver().notifyChange(newUri, null);
      return newUri;
      }
thrownew SQLException("Fail to add a new record into " + uri);
}

@Override
publicboolean onCreate() {
// TODO Auto-generated method stub
Context context = getContext();
dbHelper = new DBHelper(context);
      // permissions to be writable
database = dbHelper.getWritableDatabase();

if(database == null)
returnfalse;
else
returntrue;
}

@Override
public Cursor query(Uri uri, String[] projection, String selection,
            String[] selectionArgs, String sortOrder) {
      // TODO Auto-generated method stub
      SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
            // the TABLE_NAME to query on
           queryBuilder.setTables(TABLE_NAME);

switch (uriMatcher.match(uri)) {
      // maps all database column names
      caseFRIENDS:
                        queryBuilder.setProjectionMap(BirthMap);
      break;
      caseFRIENDS_ID:
                  queryBuilder.appendWhere( ID + "=" +
                  uri.getLastPathSegment());
                  break;
            default:
            thrownew IllegalArgumentException("Unknown URI " + uri);
```

```
                              }
            if (sortOrder == null || sortOrder == ""){
            // No sorting-> sort on names by default
               sortOrder = NAME;
                     }
 Cursor cursor = queryBuilder.query(database, projection, selection,
                               selectionArgs, null, null, sortOrder);
/**
         * register to watch a content URI for changes
   */

cursor.setNotificationUri(getContext().getContentResolver(), uri);

return cursor;
      }

@Override
publicint update(Uri uri, ContentValues values, String selection,
           String[] selectionArgs) {
      // TODO Auto-generated method stub
      int count = 0;

switch (uriMatcher.match(uri)){
caseFRIENDS:
         count = database.update(TABLE_NAME, values, selection,
selectionArgs);
break;
caseFRIENDS_ID:
         count = database.update(TABLE_NAME, values, ID +
" = " + uri.getLastPathSegment() +
                    (!TextUtils.isEmpty(selection) ? " AND (" +
                                   selection + ')' : ""),
                selectionArgs);
break;
default:
thrownew IllegalArgumentException("Unsupported URI " + uri );
    }

getContext().getContentResolver().notifyChange(uri, null);
return count;
      }

}
```

4) Register the Content Provider

We should register our provider, so the application could read from and write to our provider. To set the permissions, it is needed to add the provider to the AndroidManifest.xml file using <provider> … </provider> tags.

To add our provider, open AndroidManifest.xml file, at the last tab – named "AndroidManifest.xml" – and paste the following code.

```
<providerandroid:name="BirthProvider"
android:authorities="com.androidbook.contentex3.birthday"></provider>
```

5) When an application wants to access the data of a ContentProvider, it makes a request. These requests are handled by the **ContentResolver** object, which communicates with the ContentProvider as a client.

In our example, we add three functions (deleteAllBirthdays (View view), addBirthday(View view), showAllBirthdays(View view)) to have user interaction with the application. The user can add a new record in our provider, delete or show all of them.

Open `src/com.javacodegeeks.android.contentprovidertest/MainActivity.java` and paste the following code.

```
package com.androidbook.contentex3;

import android.app.Activity;

import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
importandroid.view.Menu;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

publicclass Contentex3Activity extends Activity {
/** Called when the activity is first created. */
@Override
publicvoid onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
```

```
    }


publicvoid deleteAllBirthdays (View view) {
// delete all the records and the table of the database provider
      String URL =
"content://com.androidbook.contentex3.birthday/friends";
      Uri friends = Uri.parse(URL);
int count = getContentResolver().delete(friends, null, null);
    String countNum = "androidbook: "+ count +" records are deleted.";
    Toast.makeText(getBaseContext(),
                countNum, Toast.LENGTH_LONG).show();

               }
publicvoid addBirthday(View view) {
// Add a new birthday record
         ContentValues values = new ContentValues();

         values.put(BirthProvider.NAME,

           ((EditText)findViewById(R.id.name)).getText().toString());

        values.put(BirthProvider.BIRTHDAY,

           ((EditText)findViewById(R.id.birthday)).getText().toString()
           );

         Uri uri = getContentResolver().insert(
                     BirthProvider.CONTENT_URI, values);

         Toast.makeText(getBaseContext(),
                       "androidbook:" + uri.toString() + " inserted!",
                    Toast.LENGTH_LONG).show();
                }

publicvoid showAllBirthdays(View view) {
           // Show all the birthdays sorted by friend's name
   String URL =
"content://com.androidbook.contentex3.birthday/friends";
   Uri friends = Uri.parse(URL);
   Cursor c = getContentResolver().query(friends, null, null, null,
                                         "name");
   String result = "androidbook Results:";

if (!c.moveToFirst()) {
```

```
            Toast.makeText(this, result+" no content yet!",
                    Toast.LENGTH_LONG).show();
                  }
else{
do{
        result = result + "\n" +
           c.getString(c.getColumnIndex(BirthProvider.NAME)) +
                              " with id " +
           c.getString(c.getColumnIndex(BirthProvider.ID)) +
                              " has birthday: " +
           c.getString(c.getColumnIndex(BirthProvider.BIRTHDAY));
                           } while (c.moveToNext());
                           Toast.makeText(this, result,
Toast.LENGTH_LONG).show();
                      }

                  }

}
```